Ingegneria del Software

Appunti di Andrea Simonetti

Modulo 1

Introduzione

Prodotto software

Codice che svolge funzione prestabilita a date prestazioni. Strutture dati per trattare informazioni. Documenti (manuale). Può essere per un cliente o il mercato, può far parte di un sistema di hardware e software. Si sviluppa, non si fabbrica o consuma.

Ingegneria del software

Suddivisione in sottoprogetti (moduli), parti già pronte, standardizzazione.

Processo di sviluppo

Insieme di attività che regolano lo sviluppo, influenza qualità, costo, tempi di realizzazione. Stabilito un ordine di esecuzione, specifica gli elaborati da fornire, assegna attività, fornisce criteri di monitoring e planning. STUDIO DI FATTIBILITÀ, ANALISI DEI REQUISITI, ANALISI DEL PROBLEMA, PROGETTAZIONE, REALIZZAZIONE E COLLAUDO MODULI, IMPLEMENTAZIONE E COLLAUDO SISTEMA, INSTALLAZIONE E TRAINING, UTILIZZO E MANUTENZIONE.

Studio di fattibilità

Valutazione di costi e benefici: alternative, scelte ragionevoli, risorse. Il risultato è un documento che definisca il problema, gli scenari per le strategie, i costi. Il committente può allocare risorse per uno studio completo.

Analisi dei requisiti

Un requisito è la descrizione di un comportamento del sistema (FUNZIONALE) o di un vincolo sul comportamento del sistema o sullo sviluppo del sistema (NON FUNZIONALE). Si stabiliscono questi e gli obiettivi consultando gli utenti.

Specifica dei requisiti

Formalizzare i requisiti, poi scrivere un documento SPECIFICA DEI REQUISITI SOFTWARE (SRS) dove si elencano i requisiti del sistema, che sia completo, non ambiguo e comprensibile a cliente e sviluppatore. Dopo averlo redatto, le specifiche vanno riviste per essere convalidate. NON SI SPECIFICA COME REALIZZARE LE FUNZIONALITÀ, È UN RIFERIMENTO E CONTRATTO.

Analisi del problema

Attenta lettura dei requisiti, individuiamo un primo insieme di problemi. Dai problemi definiamo un'architettura logica del sistema basata sul modello del dominio. L'architettura logica deve esprimere fatti oggettivi del problema (sottosistemi, ruoli, responsabilità degli scenari). È UN MODELLO CHE DESCRIVE LA STRUTTURA, IL COMPORTAMENTO ATTESO, LE INTERAZIONI DEDOTTE DAI REQUISITI SENZA PARLARE DELLE TECNOLOGIE.

Progettazione

Definiamo l'architettura generale. La decomponiamo in uno o più eseguibili, di cui per ognuno descriviamo le funzioni che svolge e le relazioni con gli altri eseguibili. Ogni eseguibile si decompone in moduli e di questi si descrivono funzioni e relazioni. Queste sono le fasi: OBJECT BASED (dati astratti, incapsulamento), OBJECT ORIENTED (ereditarietà, polimorfismo), GENERICA RISPETTO AI TIPI (generics), GESTIONE ECCEZIONI, GESTIONE EVENTI, DESIGN PATTERN, ARCHITETTURA (repository, client-server, abstract-machine), CONTROLLO DEL SISTEMA (centralizzato, event driven), CONCORRENZA, PROGETTO INTERFACCE UTENTE, PROGETTO BASI DI DATI. Da questo viene un documento di specifiche di progetto che descrive l'architettura anche formalmente con appositi linguaggi.

Realizzazione e collaudo dei moduli

Il progetto è realizzato come insieme di moduli con il linguaggio di programmazione scelto. Si fa l'analisi statica, dinamica e il debugging, collaudo dei moduli.

Integrazione e collaudo del sistema

Si integrano tra loro i moduli e si esegue il test del sistema per assicurarsi che le specifiche siano soddisfatte. ALFA TEST: sistema rilasciato per uso interno all'organizzazione del produttore. BETA TEST: rilascio controllato a pochi utenti. Si sollecitano i programmi in condizioni limite del controllo qualità.

Installazione e training

Il sistema viene consegnato al cliente, installato e messo in funzione (deployment).

Utilizzo e manutenzione

Il sistema viene utilizzato. La maggior parte dei costi è nella manutenzione. Esistono vari tipi di manutenzione: CORRETTIVA, si correggono gli errori non coperti; ADATTIVA, aumento dei servizi forniti; PERFETTIVA, miglioramento delle caratteristiche. Le modifiche vengono apportate direttamente sui programmi. Si può reingegnerizzare (refactoring), convertire il linguaggio e reverse engineering.

Processo di sviluppo software

Bisogna pianificare, controllare e gestire: analisi dei costi e gestione del gruppo di lavoro. Bisogna anche gestire i rischi relativi ai gusti, alle risorse umane, rischi tecnologici e politici.

Qualità del software

Non deve solo funzionare, ma deve anche essere la cosa giusta (validazione) costruita nel modo giusto (verifica). La revisione può avvenire con approcci BLACK BOX o WHITE BOX. Il primo approccio verifica qualità esterne, visibili a un osservatore esterno: AFFIDABILITÀ, FACILITÀ D'USO, VELOCITÀ... Il secondo riguarda qualità interne, esaminando la struttura interna, influenzano le qualità esterne: MODULARITÀ, LEGGIBILITÀ... Se soddisfano i requisiti, il software si dice CORRETTO. Il software che si comporta in maniera accettabile in casi anomali si dice ROBUSTO (termina l'esecuzione in modo pulito, graceful degradation, cioè non sono più attive alcune funzioni). Se il software è corretto e robusto si dice AFFIDABILE. Un software è facile da usare se l'utilizzatore può imparare ad usarlo, usarlo, fornire i dati, interpretare i risultati, gestire gli errori. È EFFICIENTE se c'è un buon utilizzo delle risorse hardware. È ESTENSIBILE se si può modificare facilmente per nuovi requisiti, importante per software grandi, quindi deve essere semplice l'architettura e se divisa in moduli autonomi è più probabile che una modifica riguardi pochi moduli. È RIUSABILE se si può riutilizzare almeno in parte. È VERIFICABILE se si possono fare facilmente collaudi. È PORTABILE se si può facilmente trasferire in altre architetture. Alcune di queste caratteristiche sono in contrasto tra loro, il costo del software aumenta se viene richiesto un alto livello di una di queste.

Principi OO

Tipo di dato astratto

Sono dati e codice che opera sui dati. L'interfaccia è visibile, l'implementazione è nascosta. Lo stato dell'oggetto è accessibile tramite l'interfaccia dell'ADT. L'information hiding è il concetto teorico, l'incapsulamento è la tecnica. Per definire l'ADT bisogna definire l'interfaccia, operazioni pubbliche applicabili al singolo oggetto. Per implementarlo, si definisce una classe, attributi privati, metodi pubblici, metodi privati di accesso esclusivo agli attributi.

Information hiding

L'ADT nasconde dettagli interni. Nascondendo le scelte progettuali si proteggono altre parti dal cambiamento di tali scelte. Si minimizzano le modifiche e si incentiva il riutilizzo. Si può applicare a tutti i livelli. Si possono incapsulare gli attributi in metodi accessori. Oppure possiamo fare un ADT lista di interi e nell'implementazione mettere un array o una lista.

Oggetto

È identificabile in modo univoco, ha attributi, uno stato, un insieme di operazioni per operare sullo stato o fornire servizi, un comportamento e interagisce con altri oggetti.

Classe

La classe descrive oggetti con caratteristiche comuni, stessi attributi e operazioni. A compile time definisce l'implementazione di un tipo di dato astratto. A runtime OGNI OGGETTO È L'ISTANZA DI UNA CLASSE. Le istanze sono separate le une dalle altre, ma condividono le caratteristiche generali della classe. IN UML SI RAPPRESENZA CON UN RETTANGOLO DIVISO IN 3 SEZIONI: la prima ha NOME, eventualmente STEREOTIPO (attore, controllore, evento, tabella...), eventualmente NOME DEL PACCHETTO (package, namespace); la seconda ha gli ATTRIBUTI; la terza le OPERAZIONI (se astratte in corsivo). LA FRECCIA TRIANGOLARE VUOTA TRATTEGGIATA INDICA L'INTERFACCIA CHE LA CLASSE IMPLEMENTA, IN ALTERNATIVA IL CERCHIETTO CON IL NOME DELL'INTERFACCIA. L'OGGETTO CON UNA FRECCIA APERTA TRATTEGGIATA INDICA DI CHE CLASSE È ISTANZA SE VI È SCRITTO INSTANCEOF VERSO LA CLASSE.

Ereditarietà

La gerarchia mostra la relazione tra classi. Gli oggetti di una sottoclasse esibiscono tutti i comportamenti e le proprietà della superclasse, possono essere sostituiti liberamente (LISKOV). La sottoclasse ha caratteristiche aggiuntive o esegue in maniera diversa i compiti della superclasse a patto che dall'esterno non si veda. Gli attributi e le operazioni comuni vanno definite solo una volta, mentre le cose specifiche vanno aggiunte o ridefinite. Si semplifica la realizzazione di tipi di dato simili. Ereditarietà di estensione: la sottoclasse è compatibile con i tipi della stessa catena ereditaria, IS A (Docente è una Persona), IN UML FRECCIA CONTINUA TRIANGOLARE BIANCA VERSO L'INTERFACCIA. Ereditarietà di realizzazione: si riusa il codice definito nelle superclassi, è consentito solo in C++.

Composizione e delega

IN UML FRECCIA APERTA CONTINUA, (Finestra verso Rettangolo) per esempio una Finestra contiene un Rettangolo, quindi È COMPOSTA anche da un Rettangolo. Ha accesso diretto alle operazioni pubbliche di Rettangolo (DELEGA), le interfacce restano indipendenti. È più flessibile ed estendibile, l'associazione può avvenire a runtime. PER RELAZIONI IS A USARE L'EREDITARIETÀ, ALTRIMENTI USARE LA COMPOSIZIONE.

Polimorfismo

La stessa cosa appare in forme diverse in contesti diversi (OVERLOADING), oppure cose diverse possono apparire nella stessa forma nello stesso contesto (COERCION). CLASSIFICAZIONE CARDELLI-WAGNER: polifmorfismo per universale per inclusione OOP; universale parametrico generics; ad hoc abbiamo overloading e coercion. L'OVERRIDING ridefinisce i metodi: quelli astratti sono sicuri da ridefinire, quelli concreti sono meno sicuri. Si sfrutta il concetto di BINDING DINAMICO, tramite la VIRTUAL METHOD TABLE.

Ereditarietà semplice e multipla

Nella semplice ogni classe deriva da una sola superclasse, la struttura è ad albero. Nell'ereditarietà multipla almeno una classe deriva da due o più superclassi, si ha una struttura a reticolo con conflitti di nomi. Esistono classi overlapping e disjoint. Si può passare dall'ereditarietà multipla alla composizione e delega.

Generics

Uno o più tipi sono parametrici. Ogni classe generata da una classe generica è una classe indipendente, non esistono legami di ereditarietà.

Relazioni

L'interazione tra entità diverse è possibile tramite relazioni, bisogna modellarle. GENERALIZZAZIONE O EREDITARIETÀ (IS A), REALIZZAZIONE (IMPLEMENTS), ASSOCIAZIONE (generica, aggregazione HAS, composizione HAS SUBPART), DIPENDENZA (collaborazione USA, istanza – classe, classe – metaclasse). Abbiamo sempre un cliente e un fornitore, il cliente ha bisogno del fornitore per svolgere alcune funzionalità. L'interazione tra oggetti avviene tramite l'invio di messaggi.

Processo di sviluppo OOP

L'ANALISI OO ha come obiettivo modellare la porzione di mondo reale tramite gli oggetti di analisi, ogni classe descrive una categoria di oggetti. La PROGETTAZIONE OO modella la soluzione, si possono trasformare gli oggetti di analisi e vengono introdotti gli oggetti di progettazione (liste, dizionari, finestre, tabelle...), ogni classe descrive un tipo di dato. La PROGRAMMAZIONE OO realizza la soluzione tramite linguaggi di programmazione OO per definire le classi di oggetti e sistemi runtime per creare, usare ed eliminare le istanze di queste classi, ogni classe descrive l'implementazione di un dato.

Struttura a livelli di un'applicazione

Presentation Manager

Si occupa di gestire l'interazione con l'utente. Si tratta di interfacce grafiche, quindi le API di sistema o il linguaggio HTML. Oppure un'interfaccia a caratteri con gli emulatori. Si trova all'inizio della struttura.

Data Manager

Si occupa della gestione della persistenza, quindi abbiamo delle API di sistema per interagire con il file system, oppure si parla di API per usare DBMS relazionali. Si trova alla fine della struttura.

Presentation Logic

Si occupa di gestire l'interazione con l'utente a livello logico. Parliamo di visualizzazione delle informazioni (output), accettazione dei dati (input), prima validazione dei dati e gestione dei messaggi d'errore. Fa parte dell'applicazione.

Application Logic

Si occupa della logica dell'applicazione e il controllo dei componenti. Fa parte dell'applicazione.

Data Logic

Si occupa della gestione dei dati a livello logico, cioè della consistenza dei dati, le procedure di interazione con i file (read, write, lock, unlock), le istruzioni SQL (anche commit e rollback) e la gestione degli errori. Fa parte dell'applicazione.

Middleware

È il software che permette la comunicazione tra processi. Isolano il codice dai protocolli di comunicazione. Il PRESENTATION MIDDLEWARE può essere il browser oppure il terminale. Il DATABASE MIDDLEWARE è il software che trasferisce le richieste SQL al DBMS e i dati dal DBMS all'applicazione. L'APPLICATION MIDDLEWARE gestisce la comunicazione tra componenti della stessa applicazione e può interconnettere diversi tipi di middleware.

Introduzione al .NET

Component Object Model

È un sistema OO platform independent, si possono creare software binari. Non è un linguaggio, ma uno standard. Specifica un modello a oggetti e requisiti di programmazione per l'interazione tra oggetti e oggetti COM. Un GUID identifica univocamente un'interfaccia (IID) e si può ottenere tramite QueryInterface. AddRef è usato dai client per indicare la referenza di un oggetto COM. Release è usato dai client per indicare di aver finito l'uso dell'oggetto COM. Usano una tecnica chiama REFERENCE COUNTING per assicurare che gli oggetti rimangano in memoria fintanto che vengano usati dai client per poi essere eliminati. Un oggetto COM è responsabile della liberazione della propria memoria quando il reference counting va a 0. Ci possono essere problemi quando due o più reference counting hanno riferimenti circolari. L'ereditarietà funziona SOLO CON COMPOSIZIONE E DELEGA. La posizione di un componente è memorizzata in un registro COM, può creare problemi di deployment con versioni di COM differenti, DLL hell.

Framework .NET

Semplifica il deployment, aumenta l'affidabilità del codice, unifica il modello di programmazione, è indipendente da COM, ma è fortemente integrato con COM. È un ambiente OO, everything is an object, le classi e l'ereditarietà sono pienamente supportati. Inoltre ha un garbage collector ed è fortemente tipizzato, genera eccezioni se gli errori non sono gestiti. Si può usare qualisasi linguaggio supportato e si possono estendere le stesse classi, anche senza l'uso della composizione e delega. .NET implementa CLI, CLI e C# sono standard ECMA. La CLI permette di eseguire le applicazioni di alto livello in diversi ambienti senza riscrivere l'applicazione (formato di file, sistema di tipi comune, linguaggio intermedio). Il COMMON LANGUAGE RUNTIME è l'ambiente di esecuzione runtime di .NET (codice gestito, eseguito sotto il suo controllo). Il COMMON TYPE SYSTEM è il tipo di dato supportato da .NET, fornisce un modello di programmazione unificato. Il COMMON LANGUAGE SPECIFICATION è l'insieme delle regole che i linguaggi di programmazione devono seguire per essere interoperabili all'interno di .NET, è un sotto insieme del Common Type System. L'INTERMEDIATE LANGUAGE si ottiene dal compilatore .NET nella compilazione dei sorgenti, questo poi viene passato al compilatore JIT che produrrà codice nativo da eseguire, si hanno prestazioni e portabilità. L'ambiente di esecuzione .NET Runtime riguarda dal compilatore JIT in poi.

Assembly

Unità minima per la distribuzione e versioning, è composto da un solo file che contiene: il MANIFEST, metadati che descrivono l'assembly; il TYPE METADATA, i metadati che descrivono i tipi contenuti nell'assembly; il CODICE IL che si ottiene dai linguaggi di programmazione; le RISORSE, cioè le immagini, i suoni, le icone... Può essere composto da più file (module), nel manifest si fa riferimento a file differenti con parti della struttura. Cos'è contenuto nel manifest? L'identità (nome, versione, cultura), la lista dei file che compongono l'assembly, i riferimenti ad altri assembly, i permessi. Come sono i tipi contenuti nel manifest? Hanno un nome, la visibilità, la classe base, le interfacce, i campi, i metodi, le proprietà, gli eventi; gli attributi possono essere definiti dal compilatore, dal framework o dall'utente. Questi metadati vengono usati dal compilatore, in ambienti Rapid Application Development (per avere informazioni sulle proprietà dei componenti), in tool di analisi dei tipi e del codice, in reflection, cioè l'analisi del contenuto di un assembly. Esistono assembly privati (di un'applicazione specifica, situati nella directory dell'applicazione), assembly condivisi (usati da più applicazioni, nella Global Assembly Cache, nella cartella Windows Assembly) e assembly scaricati da URL (nella cache dei download, nella cartella Windows Assembly Download). I componenti possono essere condivisi o privati e versioni diverse dello stesso componente possono coesistere anche nello stesso processo.

Common Language Runtime

Offre vari servizi. Dal IL il Common Language Runtime implementa tecnologie come il garbage collector e si occupa di funzionalità esistenti, come l'I/O su file fungendo da mediatore con il sistema operativo. Ha il supporto ai thread, il base class library support, si occupa del marshaling per COM, ha il type checker, l'exception manager, il security engine, il debug engine, il compilatore JIT, il code manager, il già citato garbage collector e il class loader.

Garbage collector

Gestisce il ciclo di vita degli oggetti .NET. Quando non sono più referenziati li elimina. Non si basa sul reference counting, quindi sono consentiti i riferimenti circolari, è più facile allocare, ma si perde la certezza di quando le cose vengono deallocate.

Gestione delle eccezioni

Un'eccezione è un errore o un comportamento inaspettato incontrato durante l'esecuzione di un programma. Può essere generata dal codice del programma in esecuzione o dall'ambiente runtime. In CLR è un oggetto che eredita da System.Exception e garantisce una gestione degli errori uniforme, eliminando i codici di errore incompatibili tra loro. Lanciare un'eccezione è THROW, catturare un'eccezione è CATCH, eseguire il codice di uscita dal blocco controllato è FINALLY.

Common Type System

Sono i tipi supportati in .NET per tutti i linguaggi, abbiamo un modello di programmazione unificato. È progettato per linguaggi OO. I tipi sono classi, strutture, interfacce, enumerativi e delegati. Il sistema è fortemente tipizzato a compile time. C'è l'overload dei metodi a compile time. I metodi virtuali si richiamano a runtime. C'è l'ereditarietà singola di estensione e multipla di interfaccia. Ad esempio System.Object, System.Int32, System.String... Tutto è un oggetto. Esistono i tipi per riferimento (allocati sull'heap, indirizzi di memoria) e i tipi per valore (allocati sullo stack, sequenze di byte). I tipi valore sono i tipi primitivi (Int, Double, Boolean...) e i tipi dell'utente (struct, enum).

Common Language Specification

È un sottoinsieme del CTS, definisce le regole di compatibilità dei linguaggi. Definisce regole per identificatori, proprietà ed eventi, costruttori, overload, ammette interfacce multiple con metodi con lo stesso nome, non ammette puntatori un managed.

Boxing e unboxing

Il boxing accade quando un tipo valore può essere automaticamente convertito in tipo riferimento medianti un UP CAST esplicito a System.Object. L'unboxing avviene se un tipo valore boxed può tornare ad essere valore esplicito mediante un DOWN CAST. Un tipo valore boxed è un clone indipendente.

Garbage collection

Ciclo di vita di un oggetto

In ambiente OO ogni oggetto deve essere usato da un programma, ha bisogno di un'area di memoria dove memorizzare il suo stato. Si ALLOCA memoria per l'oggetto, si INIZIALIZZA la memoria per rendere utilizzabile l'oggetto, si USA l'oggetto, si ESEGUE UN CLEAN UP dello stato dell'oggetto se necessario e si LIBERA LA MEMORIA.

Inizializzazione della memoria

A ogni variabile deve essere sempre assegna un valore prima che essa venga utilizzata, è il DEFINITE ASSIGNMENT e se ne occupa il compilatore. I valori di default sono usati per i tipi valore, ma questo non riguarda le variabili d'istanza, c'è il costruttore.

Clean up dello stato

In C++ e C# c'è il distruttore (~), è unico, non ereditabile, senza overload, parametri e non può essere invocato dato che è invocato automaticamente alla distruzione dell'oggetto. In Java c'è finalize, la JVM decide quando invocare il distruttore.

Garbage collection

Le risorse di un oggetto vengono rilasciate in automatico. Si evitano gli errori per i puntatori. Lo svantaggio è che vi è incertezza su quando avviene. Le strategie possono essere il TRACING, cioè determinare quali oggetti sono raggiungibili, il REFERENCE COUNTING, l'ESCAPE ANALYSIS, cioè si spostano oggetti dallo heap allo stack, l'analisi viene fatta a compile time per capire se un oggetto allocato all'interno di una subroutine è accessibile anche dall'esterno, questo riduce il lavoro del garbage collector.

Svantaggi del reference counting

Gli svantaggi sono che se due oggetti si referenziano a vicenda il contatore non andrà mai a 0, si occupa più memoria, si riduce la velocità di operazioni sui riferimenti perché va modificato il contatore, ogni modifica deve essere atomica in multithreading, ogni operazione sui riferimenti può causare la deallocazione.

Tracing

Può accadere che un programma utilizzi l'oggetto l'ultima volta molto prima che venga eliminato. Esiste il garbage sintattico, cioè oggetti che il programma non può raggiungere ed il garbage semantico, cioè che il programma non vuole raggiungere.

Allocazione della memoria

Il Common Language Runtim riserva uno spazio di memoria contiguo di indirizzamento nell'heap e memorizza in un puntatore l'indirizzo di memoria di partenza. Quando avviene la creazione di un nuovo oggetto, il CLR calcola la dimensione dell'oggetto e aggiunge due campi di 32 o 64 bit, un puntatore alla tabella dei metodi e il SyncBlockIndex. Controlla se c'è spazio dal puntatore di partenza e se non ci fosse procede con la garbage collection o lancia un'eccezione, poi aggiorna il puntatore di partenza e invoca il costruttore. È una tecnica di allocazione diversa da quella del C e di C++.

Garbage collector

Verifica se nell'heap ci sono oggetti non più usati. Ogni applicazione ha una root, un puntatore che contiene il tipo riferimento o null. Le root possono essere globali (field statici di riferimento), locali, registri CPU che contengono l'indirizzo di un oggetti di riferimento. Gli oggetti garbage non sono raggiungibili direttamente o indirettamente dalle radici. Quando il garbage collector parte, ipotizza che TUTTI GLI OGGETTI SIANO GARBAGE. Per ogni root MARCA l'oggetto referenziato e tutti gli oggetti raggiungibili. Se un oggetto è stato incontrato lo SALTA. Dopo aver scansionato tutte le root TUTTI GLI OGGETTI NON MARCATI SONO GARBAGE. RILASCIA la memoria degli oggetti garbage, COMPATTA la memoria in uso MODIFICANDO tutti i riferimenti agli oggetti spostati, UNIFICA la memoria e imposta il NextObjPtr (il puntatore di partenza). È possibile perché si conosce il tipo oggetto e si possono usare i metadati per capire quali campi fanno riferimento ad altri oggetti.

Finalization

È compito del programmatore gestire oggetti unmanaged, vale a dire risorse come il file o la connessione. L'invocazione della finalize non è deterministico, non può essere invocato direttamente essendo privato, può essere un problema se le risorse devono essere deallocate immediatamente perché c'è il garbage collector che non garantisce il rilascio immediato. È fondamentale fare il DISPOSE (rilascio) o la CLOSE (chiusura) della risorsa, è un modo deterministico. Ricordiamoci di gestire gli errori.

Il pattern Dispose

Se T vuole offrire agli utilizzatori un clean up esplicito deve implementare l'interfaccia IDisposable con metodo void Dispose(). Se la implementa si può automatizzare la dispose con il blocco using.

Tipi in .NET

Tipi in .NET

Possono essere reference type e value type. Dal punto di vista sintattico abbiamo classi, interfacce, strutture, enumerativi, delegati e array. In .NET si concretizzano sempre in una classe. Un tipo può avere o non avere costanti, campi, metodi, costruttori, operatori, operatori di conversione, proprietà, indexer, eventi, tipi annidati.

Modificatori di visibilità

Il PRIVATE è visibile nel tipo contenitore. Il PROTECTED è visibile nel tipo contenitore e nei suoi sottotipi. L'INTERNAL è visibile nell'assembly contenitore. Il PROTECTED INTERNAL è visibile sia nel tipo contenitore che nei sottotipi, che nell'assembly contenitore. Il PUBLIC è a visibilità completa. Solo internal e public sono applicabili a tipi di livello base. Il private è il default dei dati, gli altri si applicano a costanto o a campi read-only (meglio l'accesso con proprietà). Non sono applicabili per costruttori di tipo statici e distruttori (vengono solo invocati dal CLR). I membri delle interfacce, degli enum ed il namespace sono sempre pubblici. L'implementazione esplicita dei membri delle interfacce è non modificabile, o pubblica o privata.

Regole per modificatori di visibilità

Bisogna massimizzare l'incapsulamento e minimizzare la visibilità. Facciamo INFORMATION HIDING a livello di assembly (public solo i tipi significativi concettualmente), a livello di classe (public solo cose significative, protected solo funzionalità per le classi derivate come costruttori particolari, metodi o proprietà virtuali non public) e a livello di field (sempre privati e proprietà public o protected).

Costanti

Una costante è un simbolo di un valore non cambiabile. Il tipo può essere solo un primitivo per il CLR (anche string). Il valore è determinabile a compile time. È possibile usare il const anche a variabili locali.

Field

È un data member che può contenere un valore o un riferimento. Può essere di istanza (default) o di tipo (static), read-write (default) o read only (readonly) e la sua inizializzazione avviene in definizione o nel costruttore. Esiste sempre un valore di default. La differenza con le costanti sta nel fatto che le costanti vengono iniettate nel codice, comporta la ricompilazione di sorgenti nel modello cliente fornitore, invece l'accesso al field è quello di default, per cui anche con assembly diversi occorre ricompilare solo l'assembly del fornitore.

Regole per le costanti e i field

Le costanti sono davvero immutabili nel tempo. Il nome delle costanti è con la lettera maiuscola e di solito sono pubbliche. I field iniziano con l'underscore e devono essere privati, garantendo accesso con la proprietà. I field read only adottano in base alla situazione una delle convenzioni.

Modificatori di metodi

Sono VIRTUAL, ABSTRACT, OVERRIDE, OVERRIDE SEALED (o SEALED OVERRIDE). Sono applicabili a metodi, proprietà, indexer ed eventi, non possono essere statici (sono d'istanza).

Virtual

L'implementazione di un metodo virtuale può essere modificata da un membro della classe derivata. La presenza della sovrascrittura è valutata a runtime, LATE BINDING e POLIMORFISMO. Di default i metodi non sono virtuali.

Abstract

Il metodo non contiene nessuna implementazione. È implicitamente virtuale. La dichiarazione di metodi astratti è permessa solo in classi astratte. L'implementazione di un metodo astratto verrà fornita da un metodo sovrascrivente.

Override

Fornisce una nuova implementazione del metodo ereditato da una classe base. Il metodo sovrascritto deve essere virtual, abstract o override. Non si può cambiare l'accessibilità di un metodo sovrascritto. Il modificatore SEALED impedisce a una classe derivata di sovrascrivere di nuovo il metodo.

Passaggio degli argomenti dei metodi

Possono essere passati in IN (default del C#), in IN/OUT (ref), in OUT (out). Nel primo caso l'argomento deve essere inizializzato e passato per valore, le modifiche sul valore non hanno effetto. Nel secondo caso l'argomento deve essere inizializzato e viene passato per riferimento, quindi le modifiche al valore hanno effetto. Nell'ultimo caso l'argomento può non essere inizializzato, viene passato per riferimento e le modifiche sul valore inizializzato hanno effetto sul chiamante. Nel primo caso per i tipi valore non c'è alcun effetto, per i tipi reference hanno effetto sulla copia (avviene una copia del riferimento quando si invoca il metodo) e non sul riferimento originale. Nel secondo caso le modifiche hanno effetto sui tipi valore e agiscono sul riferimento originale nei tipi riferimento. Nel terzo caso, viene passato l'indirizzo dell'oggetto o del riferimento, può non essere inizializzato, ma devono essere inizializzati nel metodo.

Regole per il passaggio

Usare il più possibile il passaggio standard per valore. Se dovessimo restituire più di un valore o se dobbiamo modificare almeno un valore che viene anche usato usiamo ref se l'oggetto è stato inizializzato o out se il metodo deve completamente inizializzare l'oggetto passato.

Variadics

Si tratta di zucchero sintattico. Se volessimo scrivere un metodo con argomenti variabili possiamo sfruttare l'overloading, ma avremmo lo svantaggio di avere un numero fissato di argomenti. Per cui si può usare params tipo[] nome.

Costruttori d'istanza

La responsabilità è inizializzare correttamente lo stato dell'oggetto appena creato. C'è sempre il costruttore di default, invoca il costruttore senza argomenti della classe base. Per le classi il costruttore senza argomenti si può definire. Nel caso delle strutture non si può definire il costruttore senza argomenti. È sempre possibile definire altri costruttori con differenti SIGNATURE e differenti visibilità.

Costruttori di tipo

La responsabilità è inizializzare correttamente lo stato comune a tutte le istanze, field statici. Si dichiara static, implicitamente private, è sempre senza argomenti, senza overloading. Può accedere esclusivamente ai membri statici della classe. Se esiste, viene invocato in automatico dal CLR, prima della creazione della prima istanza della classe e prima dell'invocazione di un metodo statico della classe. Non bisogna basare il proprio codice sull'ordine di invocazione di costruttori di tipo.

Regole per i costruttori

Vanno definiti solo se necessari, cioè se usati. Non possono essere inizializzati in linea. I costruttori d'istanza possono lanciare eccezioni da gestire all'esterno del costruttore, non si possono lanciare sui costruttori di tipo.

Interfacce

Può contenere solo metodi, proprietà, indexer ed eventi pubblici e astratti. In CLR è considerata una classe astratta non derivante da System.Object. Può essere implementata sia dai tipi valore che dai tipi riferimento. È sempre considerata un tipo riferimento. Se si effettua un cast di tipo valore a un'interfaccia avviene il boxing.

Interfaccia o classe astratta?

L'interfaccia descrive un funzionalità semplice e implementabile da oggetti anche non correlati, può ereditare da altre interfacce, non può essere istanziata, né contenere uno stato, non può contenere attributi membro e metodi statici (a parte costanti comuni), non contiene implementazione, le funzionalità delle classi concrete devono essere tutte realizzate, deve essere stabile (non ci devono essere metodi aggiunti dopo), non può gestire la creazione di istanze che la implementano, la creazione deve essere fatta dai costruttori delle classi o da una classe factory. La classe astratta descrive funzionalità anche complesse comuni a oggetti omogeneri (correlati), può ereditare da interfacce, da classi astratte o concrete (almeno 1 se esiste una classe radice e al massimo 1 se non è consentita l'ereditarietà multipla), non può essere istanziata, può contenere uno stato, può contenere attributi membro e metodi statici, può essere implementata completamente, parzialmente o per niente, le classi concrete devono realizzare tutte le funzionalità non implementate, possono fornire alternative alle funzionalità implementate, può essere modificata e si può aggiungere un'implementazione di default, così viene ereditata, può gestire la creazione di istanze delle sottoclassi e la creazione può essere effettuata come per l'interfaccia, ma anche da un metodo factory della classe astratta.

Classi e interfacce base

Object

È la radice della gerarchia dei tipi, tutto deriva da Object, quindi ogni suo metodo è disponibile in tutte le classi del sistema (GetHashCode, Equals, ToString, GetType, ReferenceEquals...). Le classi derivate possono o devono sovrascrivere alcuni metodi, come Equals, ToString, GetHashCode, Finalize.

Equals

È un metodo virtuale, restituisce un booleano: true se c'è un'uguaglianza come identità, reference equality, ma per i tipi valore c'è la value equality (uguaglianza bit a bit) e per le stringhe è carattere per carattere, per Int32 è ridefinita per motivi di prestazione. Tutte le Equals devono restituire true se si fa sullo stesso oggetto, lo stesso risultato se invocato in un'oggetto o nell'altro, se i due oggetti sono NaN, lo stesso valore finché gli oggetti referenziati non vengono referenziati, false se si applica al null, il confronto a tre ((x.Equals(y) && y.Equals(z)) è positivo solo se x.Equals(z) è positivo), non deve lanciare eccezioni. I tipi che sovrascrivono Equals devono anche sovrascrivere GetHashCode o la hashtable potrebbe non funzionare. L'overloading del metodo Equals deve restituire gli stessi risultati dell'operatore di uguaglianza.

ValueType

Uno o più campi del tipo derivato sono usati per calcolare il valore di ritorno. Se uno o più campi contengono un valore variabile, il valore di ritorno è imprevedibile ed inusabile come chiave per una hash table. Bisogna considerare di scrivere un GetHashCode che rappresenti meglio il concetto di hash code per quel tipo. L'implementazione di default di Equals usa la reflection per comparare i campi corrispettivi di un oggetto e la sua istanza. Si può sovrascrivere per migliorare le performance del metodo e rappresentare meglio, anche in questo caso, il concetto di uguaglianza per il tipo. Boolean implementa le interfacce IComparable e IConvertible oltre a ValueType. Int32 implementa, oltre alle interfacce implementate da Boolean, IFormattable.

IComparable

Confronta l'istanza con un oggetto di un altro tipo, il valore di ritorno è un intero che indica l'ordine degli oggetti (minore di 0 se this precede obj, 0 se sono uguali, maggiore di 0 se this segue obj). L'argomento deve essere dello stesso tipo della classe o si genera una ArgumentException. Oltre agli accorgimenti per la Equals, due oggetti comparati in un verso e nell'altro resituiscono numeri di segno opposto e nei confronti a tre se i segni dei numeri sono uguali, allora anche nell'ultimo confronto deve essere lo stesso (A-B, B-C, quindi A-C).

IComparer

È l'interfaccia usata ad da Array.Sort o Array.BinarySearch.

IConvertible

Fornisce metodi per convertire il valore di un'istanza in un valore equivalente CLR. Se non esiste una conversione sensata viene lanciata una InvalidCastException.

Convert

È una classe usata per effettuare le conversioni dai ValueType, sono conversioni controllate. È utile anche con una stringa da convertire in valore numerico.

Conversione di tipo

Una conversione di ampliamento si effettua quando da un tipo il valore viene convertito in un tipo di dimensione uguale o superiore, viceversa per la conversione di restrizione. Possono essere implicite, le conversioni numeriche non generano eccezioni se il tipo di destinazione contiene senza perdita di informazione il valore, gli up cast consistono nell'usare una classe derivata al posto di una classe base. Le conversioni esplicite possono generare eccezioni, come nel caso di una conversione di restrizione o il down cast. C'è il boxing (up cast), e l'unboxing (down cast).

Conversione di tipo definite dall'utente

Sono metodi statici. La parola implicit indica l'utilizzo automatico, la explicit implica un cast esplicito.

Conversioni con String

C'è il ToString per convertire in stringa, con tutti i formati e le culture accessibili. C'è il Format per elementi di formato del tipo. C'è il Parse per conversioni da stringa, NumberStyles determina lo stile permesso per i parametri stringa passati ai metodi Parse. Per le valute è specificato nel NumberFormatInfo e gli attributi di NumberStyles si indicano con l'OR inclusivo bit a bit dei flag di campo.

Double

Segue le specifiche IEEE 754, supporta lo zero negativo e positivo, infinito e not a number. Epsilon è il più piccolo Double positivo. Il metodo TryParse è analogo al parse, ma non lancia eccezioni, un booleano.

Enum

Fornisce metodi per confrontare le istanze di questa classe, convertire il valore di un'istanza nella rappresentazione a stringa, da stringa di un numero in un'istanza, creare un'istanza di enumerazione e valore specifico. Si può trattare Enum come un campo di bit.

DateTime

Rappresenta un istante nel tempo dal 1 gennaio 1 al 31 dicembre 9999 delle 23:59:59. TimeSpan rappresenta un intervallo di tempo.

String

È una stringa immutabile Unicode. Una volta creata, il suo valore non può essere modificato. I metodi che sembrano modificare una String né restituiscono una nuova. Si può modificare effettivamente con StringBuilder.

ICloneable

Supporta la clonazione, crea una nuova istanza di una classe con lo stesso stato dell'esistente. Il risultato è un nuovo oggetto copia dell'istanza corrente. Può essere una copia superficiale (SHALLOW) se vengono copiati gli oggetti di primo livello (MemberwiseClone) o una copia profonda (DEEP) dove tutti gli oggetto vengono diplicati.

IEnumerable

GetEnumerator restituisce un enumeratore che può essere usato per scorrere una collezione. Espone l'enumeratore che supporta un'iterazione sulla collezione. Permettono di leggere i dati della collezione, non possono essere usati per modificare la collezione. Reset riporta l'enumeratore allo stato iniziale, MoveNext si sposta all'elemento successivo (true se ha successo, false se ha oltrepassato l'ultimo elemento), Current restituisce l'oggetto referenziato in quel momento. Non deve essere implementata direttamente da una classe contenitore, ma da una classe separata (eventualmente annidata alla classe contenitore) che fornisce funzionalità per iterare sul contenitore. Possiamo usare più enumeratori in un contenitore. La classe contenitore deve implementare l'interfaccia IEnumerable. Se il contenitore viene modificato, gli enumeratori associati vengono invalidati.

Array

Implementa IClonable, IList (IList implementa ICollection e ICollection implementa IEnumerable). Gli array possono essere di referenze o di valori, monodimensionali e multidimensionali (frastagliati o matrici).

Delegati ed eventi

Delegati

Sono oggetti che contengono un riferimento type safe a un metodo, tramite il quale può essere invocato. Un functor è un oggetto che si comporta come una funzione. Si usa per callback, in elaborazione sincrona, cooperativa (fornisce una parte di servizio) e quindi anche per la gestione degli eventi. Si dichiara con delegate, tipo di funzione, nome e parametri, poi si invoca con la new. È possibile assegnare al delegato una lista di metodi in sequenza o in modo sincrono (si aggiunge con += e si leva con -=). Un'istanza di delegato incapsula uno o più metodi, ciascuno dei quali è indicato come entità invocabile. Per i metodi statici un'entità invocabile è un solo metodo. Per i metodi di istanza è un'istanza e un metodo su quell'istanza. Un delegato applica solo una firma del metodo, non il nome, non si preoccupa della classe dell'oggetto a cui fa riferimento, sono utili per chiamate anonime. L'invocazione con più elementi procede a invocarli in modo sincrono in ordine. Ogni metodo ha gli stessi parametri. Se le chiamate hanno parametri di riferimento si passa sempre lo stesso riferimento e le modifiche sono visibili al metodo successivo. Se le chiamate hanno parametri di output o valore di ritorno il valore dipende dall'invocazione dell'ultimo delegato in elenco. È clonabile e serializzabile. La definizione di un delegato in C# indica la creazione di una nuova classe derivata.

Boss-Worker

Modelliamo un worker che effettua un'attività di lavoro e un boss che controlla le attività. Ogni worker deve notificare al boss quando il lavoro inizia, quando è in esecuzione e quando finisce. Possiamo usare il class based callback relationship, interface based callback relationship, il pattern OBSERVER, il delegate based callback relationship, l'event based callback relationship. Scegliamo i delegati. Il delegato si pone tra un caller e opzionionalmente dei call target e agisce come interfaccia con un solo metodo.

Dai delegati agli eventi

Usare campi pubblici per la registrazione fornisce un accesso eccessivo. I client possono sovrascrivere client registrati o invocare i chiamati. Fornire metodi di registrazione pubblica abbinati al campo del delegato è la soluzione migliore, ma pesante da implementare. L'event automatizza il supporto per registration, unregistration e implementation. È possibile definire gestori di registrazione a eventi.

Eventi

Si può scatenare dall'interazione con l'utente o dalla logica del programma. L'event sender è l'oggetto che scatena l'evento. L'event receiver è l'oggetto che desidera essere notifica se l'evento si verifica. L'event handler è il metodo eseguito all'atto della notifica. Quando si verifica l'evento il sender invia il messaggio di notifica a tutti i receiver (non conosce i receiver e gli handler in genere). Il meccanismo che collega sender e reciever con handler è il delegato (invocazioni anonime).

Dichiarazione di un evento

Un evento incapsula un delegato, è necessario dichiarare il tipo di delegato. I delegati hanno due parametri: la sorgente che ha scatenato l'evento e i dati relativi all'evento. Molti eventi non generano dati, in questi casi è sufficiente usare il delegato dell'evento fornito dalla libreria di classi per gli eventi senza dati, EventHandler. Delegati ed eventi personalizzati servono solo quando un evento genera dati. La classe EventArgs viene usata quando un evento non deve passare informazioni aggiuntive ai gestori, se ce n'è bisogno si deriva una classe da EventArgs, si aggiungono i dati e si usa il delegate EventHandler<TEventArgs>. La keyword event limita la visibilità e la possibilità di utilizzo del delegato. L'evento può essere trattato come un delegato di tipo speciale, null se nessun utente si è registrato oppure può essere associato a uno o più metodi da invocare.

Invocazione di un evento

È opportuno definire un metodo protetto virtuale OnNomeEvento e invocare sempre quello. È una limitazione rispetto ai delegati, l'invocazione può avvenire solo all'interno della classe nella quale l'evento è stato dichiarato.

Utilizzo di un evento

Fuori dalla classe in cui è stato dichiarato l'evento è visto come un delegato con accessi molto limitati. Un cliente può solo agganciarsi all'evento aggiungendo un delegato all'evento con += o sganciarsi dall'evento rimuovendo un delegato con -=.

Agganciarsi e sganciarsi da un evento

Agganciarsi vuol dire ricevere notifiche di un evento. Bisogna definire il metodo handler da invocare con la signature dell'evento e creare un delegato dello stesso tipo dell'evento, farlo riferire al metodo e aggiungerlo alla lista dei delegati associati all'evento. Per sganciarsi, non ricevere più notifiche dell'evento, il cliente deve rimuovere il delegato dalla lista dei delegati associati. Con l'aggancio e lo sgancio dall'esterno non si possono fare altre modifiche alla lista dei delegati o ottenerla. Si possono segnalare modifiche allo stato a clienti di tali oggetti. Gli eventi sono fondamentali per classi riusabili in tanti programmi differenti (per esempio MVC e MVP).

Interfaccia utente

Creare applicazioni Windows

Tipicamente si usano almeno una classe derivata da System.Windows.Forms.Form. Di solito si usa un thread per la UI (esegue la message pump) e gli altri thread sono usati per funzionalità scollegate dalla UI.

La classe Application

Non è instanziabile, ha metodi, proprietà ed eventi statici pubblici. Si usa per gestire l'infrastruttura della finestra applicazione. Ci sono i metodi per il message pump (Run e Exit, quest'ultima informa le message pump che devono finire l'esecuzione e chiudere tutti i form dopo che i messaggi sono stati processati) e eventi application level (Idle, ApplicationExit).

Il namespace System.Windows.Forms

Contiene classi per creare applicazioni per Windows. Possono essere raggruppate in Components, Common Dialog Boxes, Controls (Form, UserControl).

Il namespace System.Drawing

Contiene oggetti grafici basilari sottoforma di classi (Graphics, Font, Brush, Pen, Icon, Bitmap...), creatori di istanze (Brushes, Pens, SystemBrushes, SystemColors, SystemIcons, Cursors) e strutture (Point, Size, Rectangle, Color...). Graphics rappresenta un foglio di disegno, può essere in memoria, basato sul form o HDC, si usa per disegnare i controlli (DrawString(), DrawImage(), FillRectangle()...).

Componenti

Si usa come termine per indicare oggetti riusabili. In .NET un Component soddisfa questo e fornisce supporto design time. Può essere usato per il RAD (Rapid Application Development), può essere aggiunto alla toolbox di Visual Studio, trascinato nel form e manipolato su una superficie di progettazione. Il supporto design time è integrato nel framework .NET, non bisogna fare altro.

Common Dialog Boxes

Si possono usare per dare all'applicazione un'interfaccia consistente come aprire un o chiudere un file (OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog...). Inoltre Windows.Forms fornisce la MessageBox per visualizzare messaggi e prendere in input dati dall'utente.

Controlli

Sono die componenti che forniscono funzionalità all'interfaccia. Ad esempio le data entry (TextBox, ComboBox...), applicazioni di visualizzazione dei dati (Label, ListView, TreeView...), controlli per invocare comandi (Button, LinkLabel...), container (Panel, SplitContainer...), contenitori di componenti (ToolStrip, MenuStrip...). Si aggiungono al container e si possono impostare le proprietà dell'estetica e del comportamento. Si possono definire e registrare metodi per gestire eventi dell'interfaccia grafica (clic del Button, selezione del Menu, movimenti del mouse...), i comportamenti di default sono implementati dalla classe base.

La classe System.Windows.Forms.Control

È la classe base di tutti i controlli e i form. Deriva da Component e fornisce le funzionalità base ad ogni controllo, wrappa la gestione delle finestre del sistema operativo. Implementa proprietà come Size, BackColor, ContextMenu, metodi come Show(), Hide(), Invalidate(), eventi per registrazioni esterne per la notifica di eventi come Click, DragDrop, ControlAdded. Le classi derivate sovrascrivono e specializzano queste funzionalità.

La classe System.Windows.Forms.Form

È una versione specializzata di Contorl che implementa una finestra o una finestra di dialogo. Tante funzionalità sono prese dalle classi base. È specializzato nel contenere una barra del titolo, il menù di sistema, le icone di massimizzazione o minimizzazione, gestisce finestre di dialogo e implementa MDI (Multiple Document Interface). Le applicazioni derivano da Form per creare finestre e finestre di dialogo.

Controlli personalizzati

Si possono sovrascrivere i metodi virtuali per gestire la GUI (OnPaint(), OnMouseMove()...). Non occorre sovrascrivere quando la funzionalità di default va bene. Quando si sovrascrive il metodo virtuale bisogna chiamare la sua implementazione base.

Multiple Document Interface

Permette di inserire dei form figli nel form.

Altri componenti

Il namespace System.Windows.Forms fornisce classi che non derivano da Control ma forniscono lo stesso caratteristiche grafiche per l'applicazione. Le classi ToolTip ed ErrorProvider forniscono informazioni all'utente. Le classi Help e HelpProvider permettono di mostrare all'utente informazioni d'aiuto dell'applicazione.

Metadati e introspezione

Metadati

Sono dati che definiscono e descrivono altri dati. Se un componente ha abbastanza informazioni da potersi descrivere da solo, le interfacce supportate dal componente possono essere esplorate dinamicamente. Ad esempio un header in C++ è un metadato. COM e Common Object Request Broker Architecture (CORBA) usano l'Interface Definition Language per fornire metadati. È un requisito aggiuntivo da comprendere per gli sviluppatori. Sono ospitati in file separati dal tipo che lo descrivono. I metadati sono generati dalla definizione di tipo, memorizzati con essa e disponibili a tempo di esecuzione, consente la reflection.

Reflection

Può essere utilizzata per esaminare i dettagli di un assembly, per istanziare oggetti e chiamare metodi conosciuti a tempo di esecuzione e per creare, compilare ed eseguire assembly al volo.

System.Type

È il punto nodale per la reflection. Tutti gli oggetti sono istanze di tipi, si può scoprire il tipo di un oggetto con GetType o typeof. Da Object eredita MemberInfo che eredita MethodBase che eredita MethodInfo, è System.Refection e si trova in mscorlib.dll.

Very late binding

È possibile creare istanze di tipi o accedere ai membri in modo very late bound. Si istanza il tipo in memoria, si possono invocare metodi, si possono invocare accessori e modificatori di proprietà, si può sempre accedere ai metodi pubblici, si può accedere a quelli privati solo con l'autorizzazione.

System.Activator

Crea istanze dinamiche è come il late bound dell'operatore new. Alloca spazio per la nuova istanza, invoca il costruttore e restituisce un riferimento a un oggetto generico.

Attributi personalizzati

Sono un modo semplice per aggiungere informazioni ai metadati per qualsiasi elemento dell'applicazione. Possono essere utilizzati per dare ai clienti automaticamente delle funzionalità. Sono supportati in qualsiasi linguaggio .NET. Sono classi comuni che derivano da System.Attribute. Si dichiarano estendendo la classe System.Attribute, si dichiarano i costruttori e le proprietà. Si applica AttributeUsageAttribute opzionalmente, che specifica il target (a quali elementi l'attributo è applicabile), se può essere ereditato, se ci possono essere più istanze di un attributo. In C# si usano le parentesi quadre per definire il target e i parametri sono passati per posizione o per nome. Quando sono creati gli attributi, vi si può accedere con la Reflection, si può chiamare il metodo GetCustomAttributes, inherit specifica se cercare nella catena di ereditarietà, X è un'istanza di Assembly, Module, MemeberInfo e ParameterInfo.

Metaprogrammazione

La reificazione permette di preservare le informazioni di compile time per il runtime. Un sistema che permette di manipolare informazioni dinamicamente per creare nuove classi e istanziarle permette la metaprogrammazione. Diverse classi funzionano in .NET per arrivare a questo obiettivo. Si possono creare assembly al volo, Reflection.Emit permette di scrivere l'IL necessario per creare e compilare l'assembly, questo si può richiamare dal programma che lo ha creato e può essere memorizzato sul disco, così altri programmi lo possono usare. AssemblyName descrive l'identità univoca di un assembly, AssemblyBuilder rappresenta un assembly dinamico, ModuleBuilder rappresenta un modulo, TypeBuilder crea nuove istanze a tempo di esecuzione, MethodBuilder rappresenta un metodo o costruttore di una classe dinamica, ILGenerator genera istruzioni del Microsoft intermediate language.

Principi di design

Rigidità del software

Un software è rigido quando è difficile da modificare. Accade quando ogni modifica provoca una cascata di modifiche successive nei moduli dipendenti. Questo comporta la paura dei manager nel permettere la risoluzione di problemi non tecnici, non sanno quando gli sviluppatori termineranno le modifiche.

Fragilità del software

È fragile quando i cambiamenti causano comportamenti inaspettati in altre parti del sistema. Accade quando ogni correzione peggiora le cose e introduce più problemi di quelli risolti, non è manutenibile. Ogni volta che una modifica è approvata, c'è la paura di rendere il software inutilizzabile.

Immobilità del software

Non si può riusare il software in altri progetti o altre parti dello stesso software. Magari si ha bisogno di un modulo simile che però ha molte dipendenze ed il rischio o il lavoro per separare ciò che serve è enorme. Così il software viene riscritto.

Viscosità del software

La tendenza a incoraggiare modifiche che rompono il design si chiama viscosità. Con la viscosità nel design i metodi nel design sono meno usabili degli hack. La viscosità nell'ambiente rende lento e inefficiente lo sviluppo. È facile fare la cosa sbagliata, ma non quella giusta. Questo compromette la manutenibilità.

Motivi di risultati di progettazione scadenti

Un design pulito può degenerare nel corso degli anni (putrefazione). I requisiti possono cambiare in modi non previsti. Le dipendenze tra moduli non pianificate si insinuano.

Modifiche ai requisiti

I requisiti cambiano, il documento dei requisiti è il più volatile. Se i progetti falliscono per questo motivo, la colpa è della nostra progettazione. Dobbiamo trovare un modo per rendere i progetti resistenti ai cambiamenti e proteggerli dalla putrefazione.

Gestione delle dipendenze

I concetti visti sono causati da dipendenze improprie tra i moduli. È l'architettura delle dipendenze che si sta degradando con la capacità del software di essere mantenuto. Per prevenire questo, bisogna gestire le dipendenze dei moduli di un'applicazione. La progettazione OO è piena di principi e tecniche per la gestione dei moduli.

Principio zero

È il rasoio di Occam, non bisogna introdurre concetti non necessari. Quello che non c'è non si rompe. Bisogna preferire soluzioni che introducono meno ipotesi e meno concetti.

Semplicità e semplicismo

Non bisogna aggiungere complessità arbitraria al sistema software. Essa richiede uno sforzo e di solito le soluzioni semplici vengono alla fine. Non bisogna essere semplicistici, cioè levare più del necessario.

Divide et impera

È fondamentale decomporre il problema per garantire maggiore controllo e gestione. La qualità della progettazione dipende dalla qualità della decomposizione. Bisogna minimizzare il grado di accoppiamento tra i moduli del sistema (interazione fra moduli, eliminare tutti i riferimenti).

Rendere privati tutti i dati degli oggetti

Aprire il modulo vuol dire rischiare modifiche impreviste ed errori incomprensibili.

Single Responsability Principle

Se una classe ha più di una responsabilità, diventano accoppiate. Le modifiche a una responsabilità possono compromettere la capacità di realizzare le altre. Questo porta a design fragili che si rompono in modi inaspettati quando modificati. Ad esempio se la classe Rectangle viene usata per calcolare l'area da un'applicazione e disegnare sé stesso in un'altra, è meglio estrarre una classe spostando i campi e i metodi opportuni dalla vecchia classe a un'altra.

Dependency Inversion Principle

Ogni dipendenza dovrebbe puntare a un'interfaccia o una classe astratta, non a classi concrete. I moduli di alto livello (clienti) non dovrebbero dipendere dai moduli di basso livello (fornitori), entrambi dovrebbero dipendere da astrazioni. I moduli di basso livello sono più soggetti a cambiamenti. Se i moduli di alto livello dipendono da essi i cambiamenti portano a intervenire su un grande numero di moduli (RIGIDITÀ), introducono errori in altre parti del sistema (FRAGILITÀ) e i moduli di alto livello non si possono riusare perché non si riescono a separare da quelli di basso livello (IMMOBILITÀ). Funziona perché le astrazioni contengono pochissimo codice e sono poco soggette a cambiamenti. I moduli non astratti sono soggetti a cambiamenti, ma sono sicuri perché nessuno dipende da quei moduli. I dettagli del sistema sono isolati e separati da un muro di astrazioni stabili (DESIGN FOR CHANGE). I singoli moduli sono riusabili perché disaccoppiati tra loro (DESIGN FOR REUSE). Devono essere eliminate le dipendenze transitive, anche le dipendenze cicliche. LE ASTRAZIONI NON DEVONO ESSERE MODIFICATE, ma devono essere estese.

Interface Segregation Principle

Molte interfacce per un cliente sono meglio di un'unica interfaccia general purpouse. I clienti non devono dipendere da servizi che non utilizzano. Le fat interface creano un accoppiamento indiretto inutile tra clienti, se un cliente chiede una nuova funzionalità, anche tutti gli altri la avranno. È difficile la manutenzione. Se si condividono servizi in gruppi e ogni gruppo viene usato da diversi clienti si creano interfacce specifiche per ogni tipo di cliente e si implementano tutte le interfacce nella classe.

Open/Closed Principle

È importantissimo per la progettazione di entità riutilizzabili. Devono essere aperte alle estensioni e chiuse alle modifiche (interfaccia ben definita, pubblica e stabile). Vogliamo cambiare quello che fanno i moduli senza cambiare il codice dei moduli. Un modulo immutabile può esibire un comportamento che varia nel tempo con le ASTRAZIONI, il modulo deve essere immutabile, ma rappresenta tutti i possibili comportamenti. Bisogna usare le interfacce, perché da esse ci sono tantissime classi concrete che realizzano i comportamenti. Un modulo che usa astrazioni non dovrà mai essere modificato e potrà cambiare comportamento (ad esempio un Client che fa riferimento a un Server può usare l'interfaccia IServer per cambiare il comportamento richiesto). Se segue questo principio si possono aggiungere funzionalità aggiungendo nuovo codice senza cambiare codice funzionante e quello che funziona non è esposto a rotture.

Principio di sostituzione di Liskov

Il cliente di una classe deve continuare a funzionare correttamente se gli viene passato un sottotipo di tale classe base. Se un cliente usa istanze di una classe A deve poter usare qualsiasi sottoclasse di A SENZA ACCORGERSI DELLA DIFFERENZA. Il principio Open/Closed si basa sull'uso di classi concrete derivate da un'astrazione. Il principio di Liskov serve per creare classi concrete mediante l'ereditarietà. Si viola quando facciamo il refactoring di metodi virtuali delle classi derivate, bisogna riporre la massima attenzione. Per evitare le violazioni al principio di Liskov bisogna ricorrere al DESIGN BY CONTRACT.

Design by contract

Ogni metodo ha delle PRECONDIZIONI (requisiti minimi per l'esecuzione corretta di un metodo) e delle POSTCONDIZIONI (requisiti soddisfatti co l'esecuzione corretta). Questi insiemi formano un contratto tra chi invoca il metodo (il cliente) e il metodo stesso. LE PRECONDIZIONI VINCOLANO IL CHIAMANTE e LE POSTCONDIZIONI VINCOLANO IL METODO. Se sono garantite le precondizioni sono garantite le postcondizioni dal metodo. Le precondizioni devono essere poco stringenti, le postcondizioni devono essere molto stringenti, in questo modo il cliente non viola il principio di Liskov perché conosce poco sulla classe derivata. Si può modellare Square come sottoclasse di Rectangle, lo è in tutti i casi (una relazione IsA), ma ci sono problemi di implementazione. Modificando la lunghezza di un quadrato derivato da un rettangolo non si modifica l'altezza, c'è la violazione del principio di Liskov, il metodo di modifica non funziona per i sottotipi di Rectangle, perché i metodi di impostazione non sono stati dichiarati virtuali. Quando siamo obbligati a modificare la classe base il design è difettoso, viola l'Open/Close Principle. L'unica ragione per cui ha un senso rendere virtuali i metodi di altezza e lunghezza è perché esistono i quadrati, non va bene. UN MODELLO CONSISTENTE NON È NECESSARIAMENTE CONSISTENTE CON TUTTI GLI UTENTI. Ad esempio scalando un rettangolo (moltiplicando l'altezza e la lunghezza per il fattore) il quadrato verrebbe scalato due volte. Il problema non è nella funzione che scala, perché si è pensato che scalando un rettangolo allo stesso modo si scali un quadrato, l'implementazione viola il principio di Liskov. Un quadrato sarà un rettangolo, ma uno Square non è un Rectangle: il comportamento di Square non è consistente con il comportamento di Rectangle. In OOD la relazione IsA RIGUARDA IL COMPORTAMENTO PUBBLICO ESTRINSECO. Nei rettangoli i lati vengono scalati in maniera indipendente, nel quadrato no, questa indipendenza è un comportamento pubblico estrinseco. I due principi valgono se tutti i sottotipi sono conformi ai comportamenti che i clienti si aspettano dalle classi base che utilizzano. QUANDO SI DEFINISCE UNA RUOTINE SI PUÒ SOSTITUIRE CON UNA PRECONDIZIONE PIÙ DEBOLE E UNA POSTCONDIZIONE PIÙ FORTE, cioè l'utente deve conoscere solo le precondizioni e le postcondizioni di una classe base. Le classi derivate non devono aspettarsi che gli utenti obbediscano a precondizioni più forti, devono accettare tutto ciò che la classe base può accettare e le postcondizioni devono essere conformi a tutte le postcondizioni della classe base, i loro comportamenti non devono violare i vincoli stabiliti dalla classe base. Se il contratto di Rectangle è

che altezza e larghezza siano indipendenti, Square viola il contratto di Rectangle. Con il codice di test possiamo avere un'idea di come funzioni il contratto della classe.

Release/Reuse Equivalency Principle

Un elemento riusabile non può essere riusato a meno che non sia gestito da un sistema di rilascio di qualche tipo. I clienti dovrebbero rifiutare di riusare un elemento se l'autore non tiene traccia dei numeri di versione e mantiene le versioni precedenti per un po' di tempo. I pacchetti sono l'unità di riutilizzo. Gli architetti farebbero bene a raggruppare classi riutilizzabili insieme.

Common Closure Principle

Il lavoro per gestire, testare e rilasciare un pacchetto in un grande sistema non è banale. Più pacchetti cambiano, maggiore è il lavoro. Vogliamo ridurre al minimo il numero di pacchetti che vengono modificati in un ciclo di rilascio, raggruppiamo insieme classi che pensiamo che cambieranno insieme.

Common Reuse Principle

Una dipendenza da un package è una dipendenza da tutto ciò che è contenuto nel package. Se cambia e il numero di rilascio viene aggiornato tutti i client devono verificare di funzionare con il nuovo package, anche se nulla di quello che usano del package è cambiato. Le classi che non vengono usate insieme non dovrebbero essere raggruppare insieme.

Architettura dei package

I principi non possono essere soddisfatti contemporaneamente. Il Reuse Equivalency Principle e il Common Reuse Principle semplificano la vita ai riutilizzatori, l'altro ai manutentori. Il terzo rende i package più grandi. Il secondo rende i package più piccoli. Gli architetti possono impostare una struttura con cui il terzo domini per facilità di sviluppo e manutenzione. Quando l'architettura diventa stabile gli architetti possono rifattorizzare la struttura dei package per massimizzare gli altri due principi illustrati.

Acyclic Dependencies Principle

Quando si modifica un package gli sviluppatori possono rilasciare il package al resto del progetto, dopo aver verificato che il package funzioni compilandolo e collegandolo a tutti i package da cui dipende. Una singola dipendenza ciclica che sfugge al controllo può rendere l'elenco delle dipendenze molto lungo. Qualcuno deve osservare la struttura delle dipendenze e interrompere i cicli se compaiono. Un ciclo si rompe inframezzando un nuovo package e aggiungendo una nuova interfaccia.

Stable Dependencies Principle

I design non possono essere completamente statici, un po' di volatilità serve per la manutenzione del progetto. Per questo ci conformiamo al Common Closure Principle: alcuni package sono progettati per essere volatili; un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per riconciliare qualsiasi modifica con tutti i pacchetti dipendenti. La stabilità è relativa quanto lavoro serve per apportare una modifica. Un pacchetto con molte dipendenze in entrata è molto stabile perché richiede molto lavoro per conciliare le modifiche con tutti i pacchetti dipendenti.

Discussione dei principi

I package in alto sono istabili e flessibili. I package in basso sono difficili da modificare. Quelli più stabili (nella parte inferiore del grafo) sono più difficili da modificare, ma secondo il principio Open/Closed devono essere facili da estendere e si può creare un'applicazione da package instabili facili da modificare e package stabili in cui è semplice l'estensione.

Design pattern

Intoduzione

Ogni pattern descrive un problema noto e descrive il nocciolo di una soluzione, si può usare questa soluzione milioni di volte senza mai fare la stessa cosa due volte. Servono per risolvere problemi progettuali specifici e per rendere i progetti OO più riutilizzabili. Ogni design pattern formalizza l'esperienza acquisita nel risolvere un problema per riusarla in casi simili. Abbiamo quattro elementi essenziali: il NOME, identifica il pattern; il PROBLEMA, quando applicare il pattern; la SOLUZIONE, descrive il pattern negli elementi, nelle relazioni, nelle responsabilità e nelle collaborazioni; le CONSEGUENZE, descrivono i vantaggi e svantaggi nell'applicare un pattern. I nomi sono importanti perché supportano il chunking, cioè fissano il concetto e facilitano la comunicazione tra progettisti.

Classificazione

I pattern di CREAZIONE risolvono problemi inerenti alla creazione di oggetti (Factory Method, Singleton). I pattern STRUTTURALI risolvono problemi della composizione di classi e oggetti (Adapter, Composite, Decorator, Flyweight). I pattern COMPORTAMENTALI risolvono problemi sulle modalità di interazione e distribuzione delle responsabilità (Iterator, Observer, State, Strategy, Template Method, Visitor).

Singleton

Una classe ha una sola istanza e fornisce un punto di accesso globale all'istanza. La classe deve tenere traccia della sua sola istanza, intercettare tutte le richieste di creazione (ci deve essere solo un'istanza), fornire un modo per accedere all'istanza unica. Il singleton può implementare più interfacce, può essere specializzato ed è possibile creare nella GetInstance un'istanza specializzata che dipende dal contesto corrente.

Observer

Può capitare che modificando un soggetto vadano modificati anche degli osservatori. Questo si può codificare nel soggetto, ma richiede che esso sappia come debbano essere aggiornati gli osservatori. Gli oggetti sono accoppiati (CLOSELY COUPLED) e non possono essere riusati. Bisogna creare una relazione uno a molti lasca tra un oggetto e altri che dipendano da esso, la modifica del soggetto farà sì che gli altri si aggiornino di conseguenza (usare interfacce, Attach, Detach, Notify nel subject e Update nell'Observer).

Model View Controller

Si suddivide l'applicazione in model (ELABORAZIONE, STATO), view (OUTPUT), controller (INPUT). Il model gestisce i dati correlati, risponde alle interrogazioni sui dati, alle istruzioni di modifica dello stato, genera un evento quando lo stato cambia, registra gli oggetti interessati alla notifica dell'evento. In Java estende la classe Observable. La view gestisce l'area di visualizzazione, presenta all'utente una vista dei dati, mappa parte dei dati in oggetto visuali, visualizza tali oggetti in un dispositivo di output, si registra presso il model per ricevere l'evento di un cambiamento di stato, in Java implementa Observer. Il controller gestisce gli input dell'utente, mappa le azioni dell'utente in comandi, invia questi comandi al model o alla view che effettueranno le azioni opportune, in Java è listener. Nel Model View Presenter la view è passiva, cioè non interroga il model, passa tutto per il controller che viene chiamato presenter.

Flyweight

Descrive come condividere oggetti leggeri (a granularità molto fine) affinché il loro uso non sia costoso. È un oggetto condiviso che può essere utilizzato simultaneamente e in maniera efficiente da più clienti. Non deve essere distinguibile da un oggetto non condiviso, non deve fare ipotesi sul contesto in cui opera. Non bisogna istanziare direttamente un flyweight, ma usare solo una factory. Lo stato intrinseco non dipende dal contesto di utilizzo, può essere condiviso tra i clienti, si memorizza nel flyweight, invece lo stato estrinseco dipende dal contesto di utilizzo e non può essere condiviso tra i clienti, si passa al flyweight quando si invoca un'operazione.

Strategy

Definisce un insieme di algoritmi correlati, li incapsula in una gerarchia di classi e li rende intercambiabili. Il client referenzia l'oggetto che implementa un algoritmo, può dichiarare un'interfaccia che permetta all'oggetto Strategy di accedere ai dati del cliente. Strategy dichiara un'interfaccia comune a tutti gli algoritmi supportati. ConcreteStrategy implementa l'interfaccia e fornisce l'implementazione di un algoritmo.

Adapter

Converte l'interfaccia generale di una classe in un'interfaccia diversa che si aspetta il cliente. Permette a classi che hanno interfacce incompatibili di lavorare assieme. Si usa quando si vuole riusare una classe esistente e la sua interfaccia non è conforme a quella desiderata, è nota anche come WRAPPER. L'utente usa l'oggetto tramite l'interfaccia Target, l'adapter adatta l'interfaccia di Adaptee.

Decorator

Aggiunge responsabilità a un oggetto dinamicamente, è un'alternativa flessibile alla specializzazione. Il component (interfaccia o classe astratta) dichiara l'interfaccia di tutti gli oggetti ai quali possono essere aggiunte dinamicamente responsabilità. ConcreteComponent definisce un tipo di oggetto al quale deve essere possibile aggiungere dinamicamente responsabilità. Il decorator (classe astratta) mantiene un riferimento a un oggetto di tipo component e definisce un'interfaccia conforme all'interfaccia component. ConcreteDecorator aggiunge responsabilità al componente referenziato.

State

Localizza il comportamento specifico di uno stato e suddivide il comportamento in funzione dello stato. Le classi concrete contengono la logica di transizione da uno stato all'altro. Permette di emulare l'ereditarietà multipla.

Composite

Permette di comporre oggetti in una struttura ad albero con una gerarchia di oggetti contenitori e oggetti contenuti. Il component è una classe astratta che dichiara l'interfaccia e realizza il comportamento di default. Il client accede e manipola gli oggetti della composizione attraverso il component. La leaf descrive gli oggetti che non possono avere figli e definisce il comportamento. Il composite descrive oggetti che possono avere figli (contenitori) e definisce il comportamento di tali oggetti. Il contenitore dei figli deve essere un attributo di composite e può essere di qualsiasi tipo. Possiamo scegliere il RIFERIMENTO ESPLICITO AL PARENT: semplifica l'attraversamento e la gestione della struttura. Questo riferimento e la gestione devono essere posti nella classe Component. Deve essere invariante: tutti gli elementi che hanno come parent lo stesso componente devono essere gli unici figli di quel componente, quindi si incapsula l'assegnamento del parent nella classe Composite o nell'attributo parent della classe component. Oppure possiamo MASSIMIZZARE L'INTERFACCIA COMPONENT: il cliente deve vedere solo l'interfaccia di Component, quindi tutte le operazioni del cliente devono essere lì (ne realizza di default da ridefinire, anche senza senso per le foglie come Add e Remove). Possiamo optare per la TRASPARENZA: dichiaro tutto a livello più alto così il client tratta gli oggetti in modo uniforme, ma il cliente potrebbe fare cose senza senso come Add e Remove (di default dovrebbero lanciare un'eccezione) e quindi verificare se è possibile aggiungere figli all'oggetto su cui si vuole agire. Possiamo scegliere la SICUREZZA: tutte le operazioni vengono messo in Composite, qualasiasi invocazione sulle foglie genera un errore in fase di compilazione, ma il cliente deve conoscere e gestire due interfacce differenti, quindi dobbiamo disporre di un modo per verificare se l'oggetto in cui si vuole agire è un Composite.

Visitor

Definisce una nuova operazione da effettuare senza dover modificare le classi degli elementi coinvolti. Ad esempio si consideri la rappresentazione abstract syntax tree in cui i nodi descrivono elementi sintattici del programma. Su questo albero devono poter essere effettuate molte operazioni di topo diverso. Per l'AST usiamo il pattern Composite. In seguito vogliamo fare altre operazioni, se le distribuiamo su vari tipi di nodo abbiamo un sistema incomprensibile. Eliminiamo le singole operazioni dall'AST e raggruppiamo il codice relativo a un tipo in una classe. I nodi dell'AST devono accettare la visita delle istanze di queste classi visitor. Per aggiungere una nuova operazione serve solo creare una nuova classe. Il visitor deve dichiarare un'operazione per ogni tipo di nodo concreto. Ogni nodo deve dichiarare un'operazione per accettare un generico visitor. Un visitor è un'interfaccia o classe astratta che dichiara un metodo Visit per ogni classe di elementi concreti. Il ConcreteVisitor definisce tutti i metodi Visit, definisce globalmente l'operazione da effettuare sulla struttura e se necessario ha un proprio stato. Element (interfaccia o classe astratta) dichiara un metodo Accept che accetta un Visitor come argomento, ConcreteElement definisce il metodo Accept. ObjectStructure può essere realizzata come Composite o come normale collezione, deve poter enumerare i suoi elementi e deve dichiarare un'interfaccia che permetta a un cliente di far visitare la struttura a un visitor. Il pattern facilita l'aggiunta di nuove operazioni e ogni visitor nasconde i dettagli di come un'operazione va eseguita. Ogni visitor deve essere in grado di conoscere lo stato degli elementi a cui accede. Per ogni nuova classe ConcreteElement bisogna aggiungere un metodo Visit in tutti i visitor esistenti, quindi la gerarchia Element deve essere poco o per nulla modificabile, quindi deve essere stabile. Non è necessario che tutti gli elementi da visitare derivino da una classe comune. Durante ogni operazione un visitor può modificare il suo stato. Double dispatch: l'operazione che deve essere effettuata dipende dal tipo di due oggetti, cioè il visitor e l'elemento, quindi l'Accept è un'operazione di double dispatch.

Anti-pattern

Descrivono situazioni ricorrenti e soluzioni notoriamente dannose, non soddisfano i design principle.

Abstract Factory

C'è la necessita di creare oggetti connessi o dipendenti tra loro senza bisogno che il client specifichi il nome della classe concreta nel codice. Si vuole un sistema indipendente da come gli oggetti vengono creati, composti e rappresentati. Si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di prodotti. Si vuole che i prodotti che sono organizzati in famiglie siano vincolati ad essere utilizzati con prodotti della stessa famiglia. Quindi si definisce una classe che astragga la creazione di una famiglia di oggetti, istanze diverse costituiscono implementazioni diverse di membri di tale famiglia. La creazione dei prodotti è responsabilità delle classi ConcreteFactory. Quindi le classi concrete vengono isolate, si può cambiare in modo semplice la famiglia di prodotti utilizzata, promuove la coerenza nell'utilizzo dei prodotti, però è difficile aggiungere il supporto a nuovi prodotti visto che AbstractFactory definisce le varie tipologie di prodotti che è possibile istanziare, aggiungere una tipologia vuol dire modificare l'interfaccia della factory.

Implementazione

Progettazione di dettaglio

È necessario definire i TIPI DI DATO non definiti nel modello OOA, la NAVIGABILITÀ DELLE ASSOCIAZIONI tra classi e relativa IMPLEMENTAZIONE, STRUTTURE DATI necessarie per l'implementazione del sistema, OPERAZIONI per l'implementazione del sistema, ALGORITMI che implementano le operazioni, la VISIBILITÀ di classi, operazioni...

Navigabilità di un'associazione

È la possibilità di spostarsi da un qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione (a seconda delle molteplicità). I messaggi devono essere inviati solo nella direzione della freccia. A livello di analisi del problema, le associazioni di composizione e aggregazione hanno una direzione precisa. Detti A il contenitore e B l'oggetto contenuto è A che contiene B e non viceversa. A livello implementativo un'associazione può essere monodirezionale quando da A si deve poter accedere a B, ma non viceversa, mentre è bidirezionale se da A si deve poter accedere a B e da B si deve poter accedere velocemente ad A. Dal punto di vista imperativo, la bidirezionalità è molto efficiente, ma occorre tenere sotto controllo la consistenza delle strutture dati usate per l'implementazione.

Implementazione delle associazioni

Per le associazioni 0..1 o 1..1 bisogna aggiungere alla classe cliente un attributo membro che rappresenta il riferimento al fornitore o l'identificatore univoco dell'oggetto della classe se persistente, o il valore dell'oggetto della classe fornitore (in caso di composizione e molteplicità 1..1). Per le associazioni con molteplicità 0..n o 1..n bisogna aggiungere alla classe cliente un attributo membro che referenzia un'istanza della classe contenitore, cioè la classe le cui istanze sono collezioni di riferimenti a oggetti della classe fornitore, può essere realizzata o presa preferibilmente da una libreria.

Classi contenitore

Un contenitore è una classe le cui istanze contengono oggetti di altre classi. Se gli oggetti contenuti sono in numero fisso, è sufficiente un vettore predefinito del linguaggio. Se gli oggetti sono in numero variabile serve la classe contenitore. Degli esempi sono i vettori, gli stack, le liste, gli alberi. Le funzionalità minime di una classe contenitore sono inserire, rimuovere, trovare un oggetto nella collezione, enumerare (cioè iterare su) gli oggetti della collezione. Possiamo classificare i contenitore in come contengono gli oggetti (per riferimento o per valore) o se sono omogenei i contenuti (oggetti dello stesso tipo o no).

Contenimento per riferimento

L'oggetto esiste per conto proprio, può essere in più contenitori, quando viene inserito non viene duplicato, la distruzione del contenitore non comporta la distruzione degli oggetti contenuti.

Contenimento per valore

L'oggetto contenuto viene memorizzato nella struttura dati del contenitore, quando deve essere inserito in un altro contenitore viene duplicato, la distruzione del contenitore comporta la distruzione degli oggetti contenuti.

Contenimento di oggetti omogenei

Per implementare contenitori di oggetti omogenei sono ideali i generics. Si lascia generico il tipo di oggetti contenuti e ci si concentra sugli algoritmi di gestione della collezione oggetti.

Contenimento di oggetti eterogenei

Per implementare contenitori di oggetti eterogenei è necessario usare l'ereditarietà e sfruttare proprietà che un puntatore alla superclasse radice della gerarchia può puntare a un'istanza di una qualunque sottoclasse. La classe convertitore può essere generica, ma il tipo deve essere la superclasse radice della gerarchia.

Implementazione delle associazioni

Un modo alternativo per implementare un'associazione tra due oggetti è tramite un dizionario. È un tipo particolare di contenitore che associa l'oggetto chiave all'oggetto valore. La chiave può essere un oggetto qualsiasi e deve essere unica. Data una chiave, trova in maniera efficienta il valore associato.

Identificazione degli oggetti

Un oggetto può contenere un riferimento univoco a un altro oggetto. Com'è possibile identificare univocamente un oggetto per poterlo associare a un altro? Nel caso di strutture dati contenute nello spazio di indirizzamento dell'applicazione un oggetto può essere identificato univocamente dall'indirizzo logico di memoria. L'identificatore univoco è un attributo che al momento della creazione dell'oggetto viene inizializzato con un valore generato univocamente dal sistema, una chiave primaria di una tabella relazionale.

Modifiche per utilizzare il livello di ereditarietà supportato

Se non esistono strutture con ereditarietà multipla occorre convertirle in strutture con solo ereditarietà semplice. Si può usare composizione e delega: scegliere la più significativa delle superclassi ed ereditare esclusivamente da quella, le altre diventano possibili ruoli e vengono connesse mediante composizione. Le caratteristiche delle superclassi escluse vengono incorporate tramite composizione e delega. Oppure si può appiattire tutto in una gerarchia semplice e implementare un'interfaccia, una o più relazioni di ereditarietà si perdono e gli attributi e le operazioni corrispondenti devono essere ripetuti nelle classi specializzate.

Miglioramento delle prestazioni

Il software con le prestazioni migliori soddisfa i requisiti o le attese del cliente abbastanza velocemente pur rimanendo entro costi e tempi preventivati. Per migliorare la velocità percepita può bastare la memorizzazione di risultati intermedi, un'accurata progettazione dell'interazione con l'utente (ad esempio usando il multithreading), un traffico di messaggi molto elevato tra oggetti può richiedere cambiamenti per aumentare la velocità. La soluzione è che un oggetto possa accedere direttamente ai valori di un altro oggetto (aggirando l'incapsulamento), usando metodi inline, usando la dichiarazione friend, combinando insieme due o più classi. Questo tipo di modifica deve essere presa in considerazione solo dopo che tutti gli altri aspetti del progetto sono stati soggetti a misure e modifiche. L'unico modo per sapere se una modifica contribuirà in modo significativo a rendere il software abbastanza veloce è tramite le misure di osservazione.

Version Control System

Introduzione

È la gestione dei cambiamenti a documenti, programmi, grandi siti web ed altre informazioni. Si chiama anche revision control, source control, source code management.

Gestire il cambiamento del software

Tutti i programmi hanno versioni multiple: sono diversi rilasci di un prodotto, o variazioni per piattaforme, con un ciclo di sviluppo o anche semplicemente ogni volta che si modifica un programma.

Version Control

Il version control traccia le diverse versioni, permette alle versioni precedenti di essere riusate ed a tante versioni di esistere in maniera simultanea. Tutti fanno uso del version control, è utile (ad esempio per i backup).

Scenario di bug fix

Una versione viene rilasciata (1.0). Continua lo sviluppo interno (1.3). Viene trovato un bug nella 1.0, ma la 1.3 non è stabile, si rilascia la 1.0 con bugfix, abbiamo il BRANCHING (due linee di sviluppo). Il bug fix va applicato anche alla principale linea di sviluppo, così la prossima release lo avrà incluso (1.4), si ha il MERGING o UPDATING.

Scenario normale di sviluppo

Siamo nel mezzo di un progetto con sviluppatori A, B, C (1.5). A inizio giornata gnuno scansiona fa il CHECK OUT di una copia del codice, cioè hanno una copia in locale del codice (1.5a, 1.5b, 1.5c), questo isola le copie dai cambiamenti vulnerabili degli altri. A fine giornata tutti fanno il CHECK IN delle loro modifiche TESTATE, un CHECK IN è un merge dove le versioni in locale vengono copiate nel version control system (1.6). In molte aziende si effettuano dei test per il check in, se non passano, i cambiamenti non vengono accettati, si previene la perdita di tutto il lavoro.

Scenario del debugging

Si sviluppa un sistema software tramite tante revisioni (1.5 – 1.7). Nella 1.7 si scopre un bug nascosto, quando è stato introdotto? Con il version control si possono vedere le versioni precedenti e vedere quale revision ha introdotto il bug.

Scenario delle librerie

Stiamo costruendo software sopra una libreria di terze parti di cui abbiamo il codice (lib A). Iniziamo l'implementazione del software includendo le modifiche alla libreria (0.7). Una nuova versione della libreria è rilasciata, logicamente è il branch perché lo sviluppo della libreria è continuato a prescindere dal nostro sviluppo (lib B). Facciamo il merge della nuova libreria nella linea di codice applicando le modifiche alla nuova libreria (0.8).

Soluzioni casereccie

Gli sviluppatori semplicemente tengono copie multiple del programma e le etichettano. È inefficiente, molte copie simili devono essere mantenute. Richiede dei permessi read-write-execute garantiti a un insieme di sviluppatori. Questo aggiunge pressione a qualcuno che gestisce i permessi e che ha come scopo di non intaccare il codice.

Version Control System

Supporta il memorizzarsi del codice sorgente, indica una cronologia dei cambiamenti, ci si può lavorare in parallelo su diverse parti del software alla mano, fornisce un modo per lavorare in parallelo senza interferire, fornisce un modello di sviluppo, efficienta la produttività.

Concetti del VCS centralizzato

Progetti, repository, cartelle di lavoro, revisioni, branch, merging, conflitti.

Progetti

Un progetto è un insieme di file nel version control, non importa di che tipo.

Repository

È dove sono memorizzati i file correnti ed i dati storici. È tipicamente un server remoto affidabile. Tutti gli utenti condividono lo stesso repository. Si può anche chiamare depot.

Cartella di lavoro

È la copia locale dei file dal repository in un dato tempo. Tutto il lavoro fatto sui file del repository è prima fatto in una copia di lavoro. Ogni sviluppatore ne ha una nella propria macchina. È una sandbox, cioè un ambiente che isola i cambiamenti non testati del codice e gli esperimenti dall'ambiente di produzione. Copiare il contenuto del repository nella cartella di lavoro è chiamato CHECK OUT, quando l'utente ha finito, si può fare un COMMIT al repository, si chiama CHECK IN. Quindi si copiano i file dal repository alla cartella di lavoro. Si modifica il codice nella cartella di lavoro, si aggiorna il repository dalla cartella di lavoro (prima si verifica che il codice sia corretto) e così daccapo.

Revisioni

Consideriamo di fare il check out di un file, lo modifichiamo e ne effettuiamo il check in. Questo crea una nuova versione del file, si incrementa il minor version number. Molte modifiche sono minime. Per efficienza, non memorizziamo tutto il nuovo file, memorizziamo solo il diff, minimizza lo spazio, però potrebbe rallentare il check in e il check out (bisogna applicare i diff di tutte le precedenti versioni per computare il file corrente). Con i diff, si memorizzano anche il minor version number e altri metadati (autore, orario di check in, messaggio di log, risultati degli smoke test).

Branch

Sono due revisioni del file (due persone fanno il check out della 1.5, si ha 1.5.1 e 1.5.2). Normalmente però non si creano dei branch, le modifiche si aggiungono alla linea di sviluppo, deve essere creato esplicitamente un branch.

Lock-Modify-Unlock

Bisogna evitare che due utenti accedano contemporaneamente al repository, creino le loro modifiche, le prime vengano scritte e poi sovrascritte dalle seconde. Quindi questo modello permette a una sola persona alla volta di modificare un file. Ogni volta che qualcuno vuole modificare il file deve prima fare il lock. Se un file è bloccato, nessun altro può modificarlo. Quando le modifiche sono finite, si fa l'unlock del file. Potrebbero esserci problemi amministrativi: un utente può fare il lock su un file e dimenticarsene anche per tanto tempo, impedendo ad altri di accedere al file. Potrebbe esserci della serializzazione non necessaria: se si vogliono modificare parti diverse del file contemporaneamente senza pericoli sarebbe impossibile. Potrebbe crearsi un falso senso di sicurezza: se si fa il lock su due file che dipendono l'uno dall'altro e le modifiche li rendono inusabili il lock non previene questo problema.

Copy-Modify-Merge

Non c'è il lock, quando qualcuno fa il check in della sua copia di lavoro i cambiamenti si aggiungono a quelli degli altri utenti. Chi pubblica per prima la propria modifica viene accettata. Quella dell'altro vengono rifiutati sulla versione vecchia, dovrà riscriverli su quelli nuovi.

Conflitti

Si hanno quando due programmatori modificano la stessa parte di codice, il sistema non sa cosa deve fare, quindi segnala il conflitto e il version control lo mostra. I conflitti vanno risolti a mano. Sono sintattici, basati sulla vicinanza dei cambiamenti: vanno in conflitto i cambiamenti sulla stessa riga. La mancanza di conflitti non vuol dire che i cambiamenti degli sviluppatori possano lavorare bene assieme. Il merging è sintattico, gli errori semantici non creano conflitti, ma il codice è comunque sbagliato, saremmo fortunati se non compilasse.

LMU o CMM

CMM si esegue molto bene, gli utenti possono lavorare in parallelo senza aspettare l'uno e l'altro. Molti cambiamenti non si sovrappongono e i conflitti sono rari. L'ammontare di tempo per risolvere i conflitti è molto minore del tempo perso con il lock. LMU è ottimo per file in cui non si può fare il merge, come le immagini.

VCS centralizzati

Concurrent Version System (1986 – 2008) controlla solo i file, non le cartelle o i metadati. Subversion (SVN, 2000 -) è un progetto Apache dal 2009.

Migrare ai VCS distribuiti

Usando il CMM il repository centrale perde il suo senso. Piuttosto che un unico repository centrale, dove i clienti si sincronizzano, ogni copia di lavoro degli utenti può funzionare da repository. Il revision control distribuito porta alla sincronizzazione tramite scambi di patch.

La differenza col centralizzato

Non c'è una copia canonica del codice, solo copie funzionanti, comunque si può creare il repository ufficiale. Operazioni comuni (commit, cronologia) sono veloci, non bisogna comunicarle a un server. Ogni copia funzionante funziona come backup remoto del codice o della sua cronologia.

Albero delle revisioni

Le revisioni sono pensate per una linea di sviluppo con branch che formano un albero diretto, con anche diverse linee di sviluppo. A dire il vero, la struttura è più complicata, forma un grafo diretto aciclico, ma è conveniente definirlo “albero con i merge”.

Concetti del VCS distribuito

Trunk (unica linea di sviluppo, chiamata anche Baseline, Mainline o Master), Branch, Tag, Push/Pull.

Branch

Un insieme di file sotto il version control possono passare al vaglio del branch ad un certo punto sicché da quel punto in avanti due copie diverse di questi file vengono sviluppate a ritmi diversi o in modi diversi e indipendenti. È una copia completa del codice, la storia è inclusa. Quando gli update sono consolidati, si può fare il merge nel main.

Tag

È un’etichetta che si riferisce a importanti snapshot nel tempo, consistenti tra molti file. Questi file ad un certo punto possono essere taggati con un nome o un revision number intuitivo. Tag equivale a release.

Push/pull

Vuol dire copiare le revisioni da un repository all’altro. Il push è iniziato dal sorgente, il pull dal repository ricevente. Di solito, il contribuente fa una richiesta di pull e il maintainer deve fare il merge delle richieste di pull.

Workflow

Affinché gli altri vedano i cambiamenti apportati, devono accadere 4 cose: si fa il commit, si fa il push, si riceve il pull e si riceve l’update. Il commit e l’update fanno cambiamenti tra la copia funzionante e il repository locale. Invece il push e il pull fanno cambiamenti tra il repository locale ed i repository altrui.

Di cosa si fa il pushing o pulling?

I VCS distribuiti mettono l'accento sul condividere i cambiamenti, ognuno dei quali ha un GUID. Queste azioni cambiano set, non veri file. Ogni cambiamento è facile da tracciare, grazie al GUID.

Migliori pratiche

Usare messaggi di commit descrittivi (indica lo scopo), fare di ogni commit un'unità logica (è più semplice mappare i cambiamenti), evitare commit indiscriminati, incorporare frequentemente i cambiamenti altrui, condividere i propri cambiamenti frequentemente, coordinarsi con i colleghi, non fare commit di file generati.

Vantaggi e svantaggi del VCS distribuito

Da un lato ognuno lavora in una sandbox, funziona offline, è veloce (è tutto fatto localmente), branching e merging sono semplici, c'è bisogno di meno manutenzione. Dall'altro lato c'è sempre bisogno di un backup, non esiste un'ultima versione vera e propria, non ci sono degli autentici revision number.

VCS decentralizzati

BitKeeper, gratuito dal 2000 al 2005, dal 2005 al 2016 a pagamento, dal 2016 open source. Mercuiral (2005 -), alternativa a BitKeeper non più gratuita. Git (2005 -), creato da Linus Torvalds per il kernel Linux.

Concetti di Git

CARTELLA, il repository principale; CARTELLA DI LAVORO, una copia del repository; STAGING AREA, un file indice che specifica quali file modificati dovranno essere salvati (si farà il commit) nel repository.

Status di un file in Git

COMMITED, il file è salvato nel repository locale; MODIFICATO, è stato modificato, ma non né stato fatto il commit; STAGED, è marcato per il commit.

Workflow di Git

CHECKOUT DEL PROGETTO, dalla cartella Git alla cartella di lavoro; MODIFICA DEI FILE, le modifiche rimangono nella cartella di lavoro; STAGE DEI FILE, si selezionano i cambiamenti per lo stage, sono ammessi solo quelli che devono entrare nel prossimo commit; COMMIT, si memorizzano i file di snapshot permanentemente. Git non usa i diff, usa gli snapshot.

Modulo 2

Modelli

Modello

È una rappresentazione di un oggetto o un fenomeno reale che produce caratteristiche o comportamenti ritenuti fondamentali per il tipo di ricerca che si sta svolgendo. È un insieme di concetti e proprietà volti a catturare aspetti essenziali di un sistema, collocandosi in uno spazio preciso concettuale. È una visione semplificata di un sistema che rende sé stesso più accessibile alla comprensione e valutazione, facilita il trasferimento dell'informazione e la collaborazione tra persone.

Modelli e processi software

Nel processo di produzione del software il lavoro di diversi attori si basa su un insieme di conoscenze implicite all'interno del processo. Lo scopo di un processo basato su modello è esplicitarle, tramite diagrammi formali, con un linguaggio preciso. I diagrammi descrivono in modo coinciso e preciso conoscenze del problema, servono per individuare rischi e scelte progettuali. Inoltre, la mente umana trova difficoltà a impostare ragionamenti efficaci con dettagli minuti. I linguaggi per descrivere i modelli sono il culmine del livello di astrazione raggiunto rispetto alle macchine.

Caratteristiche di un modello

I modelli che descrivono un sistema devono fornire una descrizione completa, consistente, non troppo ridondante. La transizione tra modelli deve essere continua, i modelli devono essere connessi tra loro in modo sistematico (l'elemento di un modello deve avere il o i suoi corrispettivi in un altro).

Tracciabilità

Si deve poter mappare gli elementi di un modello in elementi di un altro, in qualsiasi direzione si percorra la sequenza di modelli generati. Si garantisce coerenza tra modelli, si crea un percorso logico che va dai requisiti al codice, si tengono sotto controllo le modifiche. È un compito difficile.

Linguaggi di modellazione

Un linguaggio di modellazione è semiformale, si usa per descrivere sistemi di qualunque natura. Un modello di sistema software si chiama MODELLO SOFTWARE. Con i diagrammi rappresentiamo il modello tramite il linguaggio. Esistono linguaggi diversi con diversi poteri espressivi (UML, XMI, OPM). Abbiamo modelli SEMANTICI DI DATI (E-R), orientati all'ELABORAZIONE DATI (diagrammi di flusso, DFD), orientati alla CLASSIFICAZIONE (OO), OPERAZIONALI (automi a stati finiti, reti di Petri), DESCRITTIVI (logica del primo ordine, logica temporale). Bisogna usare un linguaggio di modellazione per risolvere il problema di comunicazione tra progettisti, presenti e futuri, e clienti. Il linguaggio naturale è impreciso, il codice è troppo dettagliato, quindi serve un linguaggio sufficientemente PRECISO, FLASSIBILE per le descrizioni in modo da decidere quanto entrare nel dettaglio e STANDARD. Esiste lo Unified Modeling Language.

Codice

È una rappresentazione del modello molto dettagliata, dà una visione piatta, non evidenzia i punti salienti, non aiuta ad avere una visione d'insieme istantanea. Esigiamo di un modello e del codice, MA SONO DISALLINEATI. Questo accade già in implementazione, ci sono modifiche nel codice che non si riflettono sul modello e viene meno la tracciabilità. Generare modelli dal codice si porta i difetti del codice e non ha senso. Per mantenere la tracciabilità bisogna apportare modifiche al modello, generare da questo il codice e negli altri casi modificare il codice mantenendo la coerenza col modello.

Processo di sviluppo

È un insieme ordinato di passi che coinvolge tutte le attività, vincoli e risorse per produrre un output a partire dai requisiti d'ingresso. Un processo è composto da fasi in relazione le une alle altre. Ogni fase è una porzione di lavoro, identifica risorse e vincoli e può essere composta da più attività. Il processo di sviluppo software è un insieme coerente di POLTICHE, STRUTTURE ORGANIZZAZIONALI, TECNOLOGIE, PROCEDURE, DELIBERABLE necessari per concepire, sviluppare, installare, manutere un prodotto software.

Generiche fasi

La SPECIFICA dice cosa dovrebbe fare il sistema, con i vincoli; lo SVILUPPO è la produzione del sistema; la VALIDAZIONE è il test sul sistema, se è quello che il committente voleva; l'EVOLUZIONE sono i cambiamenti accordati a modifiche dei requisiti o aggiunta di funzionalità.

Modelli di processo

Sono una rappresentazione semplificata di un processo visto da una prospettiva. Prescrivono le fasi in cui è organizzato e l'interazione e la coordinazione del lavoro tra le fasi. Definiscono dei template su cui organizzare un vero processo di sviluppo.

Modello a cascata

Le fasi sono distinte, con retroazione finale: STUDIO DI FATTIBILITÀ, ANALISI, PROGETTAZIONE, IMPLEMENTAZIONE, COLLAUDO, MANUTENZIONE. Introdurre cambiamenti in fasi avanzate di sviluppo ha un costo troppo elevato, quindi ogni fase deve essere svolta in maniera esaustiva prima della prossima. I SEMILAVORATI sono le uscite di una fase e gli input della successiva (documentazione, codice, sistema). È necessario definire quali semilavorati produrre e quali date rispettare. I limiti di questo modello sono dati dalla rigidità, soprattutto l'immutabilità dell'analisi (i clienti devono sapere esattamente cosa vogliono) e l'immutabilità del progetto (è possibile sviluppare il progetto senza di aver prima scritto una riga di codice). Questi due limiti non si riscontrano con la realtà, dove i requisiti cambiano in continuazione e ci possono essere modifiche alle scelte di progetto. UN'EVOLUZIONE AGGIUNGE LA RETROAZIONE A UN LIVELLO.

Prototipo

Prima di iniziare a lavorare sul sistema si può realizzare un prototipo che fornisca agli utenti una base in cui sviluppare le specifiche. I prototipi sono usa e getta, è un approccio dispendioso che annulla i vantaggi del modello a cascata. Il prototipo è un modello approssimato dell'applicazione, deve essere sviluppato in tempi e costi minimi, mostrato al cliente per rifinire i requisiti. Se usa e getta, si deve concentrare sugli aspetti in cui i requisiti non sono chiari. POSSONO ANCHE NON ESSERE USA E GETTA, dal prototipo si passa piano piano al prodotto finale, bisogna lavorare a stretto contatto con il cliente. Prima si sviluppano le parti chiare del sistema e poi si aggiungono nuove funzionalità richieste dal cliente, è la PROGRAMMAZIONE ESPLORATIVA.

Modelli evolutivi

Da specifiche molto astratte si sviluppa un primo prototipo da sottoporre al cliente e raffinare. Ci sono diversi modelli evolutivi, ma tutti vertono sul fatto che il prototipo evolva in prodotto finito un po' alla volta. I ogni iterazione ci si può confrontare col cliente per le specifiche (raffinamento dell'analisi) e rivedere le scelte di progetto (raffinamento del design).

Extreme programming

Se le iterazioni sono brevissime si parla di extreme programming. Non è visibile il processo di sviluppo, il sistema è poco strutturato, bisogna essere abili nella programmazione (team ristretto). Va bene per sistemi di piccole dimensioni, di breve durata o parti di sistemi più grandi. È necessaria la comunicazione tra sviluppatori e tra sviluppatori e clienti. C'è più codice per i test che per il programma. Il codice deve essere molto semplice, consente il riutilizzo senza pensarci. Non bisogna avere paura del refactoring continuo.

Modelli ibridi

Sono sistemi composti di sotto-sistemi, in cui applicare il modello evolutivo (specifiche ad alto rischio) o a cascata (specifiche ben definite). E conviene raffinare prototipi funzionanti del sistema o di sue parti con un approccio incrementale-iterativo.

Sviluppo incrementale

Costruiamo il sistema sviluppando parti ben definite in sequenza. Quando una parte è completata, non va più modificata. Bisogna essere in grado di specificare perfettamente i requisiti prima della costruzione.

Sviluppo iterativo

Si effettuano molti passi del ciclo di sviluppo iterativamente aumentandone il livello di dettaglio ad ogni iterazione, non funziona bene per progetti significativi.

Sviluppo incrementale-iterativo

Si individuano sottoparti abbastanza autonome, si realizza il prototipo di una di esse, si continua con le altre parti, si aumenta pian piano l'estensione e il dettaglio dei prototipi considerando le parti interagenti e così via.

Sviluppo a componenti

Si fa l'analisi dei requisiti, analisi e progettazione OO. Dalla progettazione si ricercano i componenti da riusare e si sviluppano. Dalla progettazione si scrive un prototipo e si passa alla valutazione del prototipo, che riporta all'analisi OO. La valutazione può portare all'installazione e utilizzo che eventualmente ci ricongiunge all'analisi dei requisiti, come dice il modello evolutivo.

Rational Unified Process

È un modello ibrido, contiene modelli di processo generici, è un framework adattabile che può essere usato in progetti di diversi contesti. È pensato per progetti di grandi dimensioni. Abbiamo tre visioni del processo di svilippo: DINAMICA, mostra le fasi del modello nel tempo; STATICA, mostra attività del processo coinvolte; PRATICA, suggerisce buone prassi da seguire durante il processo.

Prospettiva dinamica

INCEPTION, generalizzazione dell'analisi di fattibilità, lo scopo è delineare il business case, cioè comprendere il mercato al quale il progetto afferisce e identificare elementi importanti perché esso conduca a un successo commerciale, identificare le entità esterne che interagiscono col sistema, si usano lo use-case diagram, la valutazione dei rischi e requisiti grossolani; ELABORATION, definisce la struttura complessiva del sistema, comprende l'analisi di dominio e una fase di progettazione dell'architettura, i casi d'uso devono essere completi all'80%, si deve descrivere l'architettura del sistema, si sviluppa un'architettura eseguibile con i principali use-case, si revisiona il business case e si pianifica il progetto complessivo e se ciò non accade il progetto può essere abbandonato perché inizia la fase più rischiosa; CONSTRUCTION, si progetta, implementa e testa il sistema, le parti sono sviluppate in parallelo e integrate per avere un sistema software funzionante con documentazione; TRANSITION, il sistema passa al cliente finale, si fanno attività di beta testing, si verifica che il prodotto sia conforme alle aspettative altrimenti si ricomincia.

Prospettiva statica

Si concentra sulle attività di workflow. Ci sono 6 workflow principali e 3 di supporto, sono orientati a modelli associati a UML. Tutti i workflow possono essere attivi in ogni stadio del processo, non c'è correlazione tra visione statica o dinamica e workflow.

Workflow

MODELLAZIONE DELLE ATTIVITÀ AZIENDALI, si usa il business case; REQUISITI, vengono identificati gli attori che interagiscono e i casi d'uso per modellare i requisiti; ANALISI E PROGETTO, viene creato e documentato un modello di progetto usando modelli architetturali, dei componenti, degli oggetti e sequenziali; IMPLEMENTAZIONE, i componenti sono implementati e strutturati nei sottosistemi, si può anche generare automaticamente il codice; TEST, è un processo eseguito in parallelo all'implementazione, il test finale segue il completamento di tutti gli altri test; RILASCIO, viene creato e distribuito un rilascio per gli utenti, da installare nella postazione di lavoro; GESTIONE DELLE MODIFICHE (DI SUPPORTO), gestisce i cambiamenti nel sistema; GESTIONE DEL PROGETTO (DI SUPPORTO), gestisce lo sviluppo del sistema; AMBIENTE (DI SUPPORTO), rende disponibili strumenti per gli sviluppatori. Le fasi possono essere eseguite ciclicamente con risultati incrementali, così come tutto l'insieme delle fasi può essere eseguito in modo incrementale.

Prospettiva pratica

Descrive le buone prassi dell'ingegneria del software. SVILUPPARE IL SOFTWARE CICLICAMENTE, pianificare gli incrementi in base alle proprietà del cliente, sviluppare e consegnare subito le funzioni con priorità più elevata; GESTIRE I REQUISITI, documentare i requisiti del cliente esplicitamente e i cambiamenti, analizzare l'impatto di quest'ultimi prima di accettarli; USARE ARCHITETTURE BASATE SUI COMPONENTI; CREARE MODELLI VISIVI DEL SOFTWARE, usando grafici UML; VERIFICARE LA QUALITÀ DEL SOFTWARE, deve raggiungere gli standard di qualità; CONTROLLARE LE MODIFICHE DEL SOFTWARE, usando un sistema per la gestione di modifiche e per la gestione della configurazione.

Analisi dei requisiti

Requisiti

Rappresentano la descrizione di servizi forniti e vincoli operativi. C'è una disciplina dedicata alla ricerca, analisi, documentazione e verifica, l'ingegneria dei requisiti (RE). I REQUISITI UTENTE dichiarano quali servizi il sistema dovrebbe fornire, i vincoli sotto cui deve operare, sono molto astratti e di alto livello, sono espressi in linguaggio naturale, con qualche diagramma; i REQUISITI DI SISTEMA definiscono le funzioni, i servizi e i vincoli operativi del sistema in modo dettagliato, è una descrizione dettagliata di quello che il sistema deve fare, il DOCUMENTO DEI REQUSITI DEL SISTEMA fa parte del contratto e deve essere preciso, deve definire esattamente cosa deve essere sviluppato. I requisiti di sistema si dividono in requisiti funzionali, non funzionali e di dominio.

Requisiti funzionali

Descrivono quello che il sistema dovrebbe fare, sono elenchi di servizi che il sistema deve offrire. Per ognuno possiamo indicare come reagire a particolari input, come comportarsi in alcune situazioni, cosa non dovrebbe fare. Le specifiche devono essere COMPLETE, cioè tutti i servizi sono definiti e COERENTI, cioè i requisiti non devono avere informazioni contraddittorie.

Requisiti non funzionali

Possono essere di tre tipi: DEL PRODOTTO, specificano proprietà del sistema (affidabilità, prestazioni, protezione dati...); ORGANIZZATIVI, possono vincolare anche il processo di sviluppo adottato (politiche e procedure di organizzazione, uso di un case tool, limiti di budget...); ESTERNI, derivano da fattori estranei al sistema (interoperabilità, legislazioni sulla privacy, requisiti etici). Possono essere difficili da verificare perché spesso sono vaghi, contrastano spesso con i requisiti funzionali, vanno studiati e analizzati con cura.

Requisiti di dominio

Derivano dal dominio di applicazione del sistema. Includono termini del dominio o che si rifanno a concetti del dominio. Sono specialistici, è difficile capire come questi si rapporti con altri requisiti. Vanno comunque analizzati con cura perché riflettono i fondamenti del dominio dell'applicazione.

Analisi dei requisiti

L'obiettivo è definire le proprietà che il sistema dovrà avere senza descriverne la realizzazione. Avremo una serie di documenti con la descrizione dettagliata dei requisiti, una base di partenza per l'analisi del problema. Per determinare al meglio i requisiti bisogna interagire molto con l'utente e conoscere l'area applicativa.

Raccolta dei requisiti

Bisogna raccogliere tutte le informazioni su cosa il sistema deve fare secondo le intenzioni del cliente. Non ci sono passi formali, dipende dal problema. Avremo un documento testuale approvato dal cliente e scritto da un'analista, con un inizio di vocabolario o glossario contenente tutti i termini senza ambiguità. Coinvolgiamo analisti, clienti e a volte esperti del dominio. Usiamo le interviste, studiamo documenti che esprimono i requisiti in forma testuale, osserviamo passivamente o attivamente il processo, studiamo sistemi software esistenti e usiamo prototipi. È complesso gestire le interviste, perché i clienti possono avere una vaga idea dei requisiti, non essere in grado di spiegarli chiaramente, possono essere irrealizzabili o in conflitto tra loro, i clienti possono essere poco disponibili a collaborare.

Validazione dei requisiti

Ogni requisito va validato e negoziato con i clienti. Si svolge in parallelo alla raccolta. Deve essere VALIDO, inerente al problema da risolvere, CONSISTENTE, non in conflitto o sovrapposto ad altri, REALIZZABILE, COMPLETO.

Documento dei requisiti

Deve specificare in modo chiaro e univoco cosa farà il sistema. I requisiti devono essere chiari, precisi, corretti, consistenti, completi, tracciabili, concisi, modificabili, non ambigui, verificabili. Inoltre deve contenere in versione iniziale il dizionario dei termini. Si possono organizzare i requisiti in una tabella (id, requisito, tipo, tipo vuol dire funzionale oppure no oppure del dominio).

Cambiamento dei requisiti

È normale che i requisiti cambino nel tempo, vanno gestiti. Più lo sviluppo è avanzato, più il cambiamento è costoso, quindi quando si cambia bisogna valutare la fattibilità, l'impatto e il costo. È consigliato sviluppare sistemi che siano il più possibile resistenti a cambiamenti di requisiti, teniamone traccia nella tabella dei requisiti.

Analisi del dominio

Bisogna definire la porzione di mondo reale rilevante per il sistema. Un principio importante è l'astrazione, permette di gestire la complessità intrinseca del mondo reale, ignoriamo gli aspetti non importanti. Abbiamo una prima versione di vocabolario basata sui sostantivi dei requisiti. Possiamo fare rifermento a sistemi della stessa area applicativa, identifichiamo entità e comportamenti comuni ai sistemi e realizziamo schemi e componenti riusabili in questi.

Analisi dei requisiti

Dobbiamo definire il comportamento del sistema da realizzare. Quindi il risultato è un modello comportamentale o dinamico che descrive in modo chiaro e coinciso le funzionalità del sistema, cosa deve fare per soddisfare il cliente. Ci sono due strategia: SCOMPOSIZIONE FUNZIONALE, identificare le singole funzionalità previste dal sistema; ASTRAZIONE PROCEDURALE, ogni operazione è un'entità, anche se realizzata da più operazioni di basso livello. La scomposizione in funzioni è volatile perché i requisiti cambiano. Vanno realizzati tutti i requisiti della tabella dei requisiti, facciamo attenzione ai sostantivi e ai verbi del testo della specifica dei requisiti, si formalizza dai sostantivi il primo modello del dominio, dai verbi il modello dei casi d'uso, cioè l'insieme delle azioni che il sistema dovrà compiere, dall'analisi sorgono sempre nuovi requisiti e quindi bisogna aggiornare la tabella relativa.

Vocabolario

Nella modellazione è importante usare la specifica terminologia, specificata nel vocabolario. Migliora la comunicazione tra gli attori del processo di sviluppo (analisti e progettisti). Ogni entità si evince dai requisiti e può essere espressa come UML e messa in relazione con le altre entità per creare il primo modello del dominio.

Analisi e gestione rischi

Analisi sistematica e completa di tutti i possibili rischi che possono far fallire o intralciare la realizzazione del sistema in qualsiasi momento. Ogni rischio ha una probabilità che avvenga (se è al 100% è un vincolo) e un costo (quali sono gli effetti indesiderati o le perdite). RISCHI DEI REQUISITI, il sistema non soddisfa le esigenze del cliente; DELLE RISORSE UMANE, non hanno la giusta esperienza per il progetto; DELLA PROTEZIONE E PRIVACY, quali dati trattare e come proteggerli da attacchi informatici; TECNOLOGICI, se la tecnologia è corretta e quali saranno i suoi futuri sviluppi; POLITICI, se ci sono forze politiche che intralciano il progetto. Ci sono due strategia: REATTIVA, ai problemi si trova una soluzione; PREVENTIVA, si individuano i rischi e li si classifica per importanza, poi si predispone un piano per reagire in modo controllato ed efficace.

Casi d'uso e scenari

Permettono di formalizzare i requisiti funzionali, aiutano a comprendere il funzionamento del sistema, e permettono di comunicare meglio con il cliente. Tutti i casi d'uso sono l'immagine del sistema verso l'esterno. Si individua il CONFINE del sistema; si individuano gli ATTORI, con ruoli di utente nei confronti del sistema; si individuano i CASI D'USO, i servizi richiesti al sistema da un attore o da un altro caso d'uso; si disegnano i diagrammi dei casi d'uso, modellano le associazioni tra gli attori e i casi d'uso e tra vari casi d'uso; si descrivono i dettagli di ogni caso d'uso, sia l'interazione tra attore e sistema che le elaborazioni per soddisfare la richiesta; si ricontrollano e validano con l'utente. Un caso d'uso viene sempre avviato dall'intervento diretto o indiretto di un attore, si conclude con successo se l'obiettivo viene raggiunto, con fallimento se l'obiettivo non viene raggiunto. Viene sempre descritto da nessuna o più precondizioni (condizioni verificate prima dell'esecuzione del caso d'uso), da almeno uno scenario (sequenze di passi che descrivono le interazioni tra l'attore e il sistema per raggiungere l'obiettivo, con eventuali ramificazioni), nessuna o più postcondizioni (condizioni vere se il caso d'uso viene eseguito con successo). Ogni sequenza di passi deve usare il vocabolario di dominio e seguire una forma narrativa strutturata, così il committente potrà comprenderli, validarli e partecipare attivamente alla definizione. Un caso d'uso comprende uno scenario principale ed eventualmente scenari alternativi (varianti anomale del flusso, che scattano da opzioni o condizioni d'errore...). GENERALIZZAZIONE / SPECIALIZZAZIONE, si ha quando un caso d'uso è simile a un altro ma fa qualcosa di più, anche un attore può essere la specializzazione di un altro attore; INCLUSIONE, quando un caso d'uso usa almeno una volta un altro caso d'uso; ESTENSIONE, quando è necessario aggiungere un comportamento opzionale a un caso esistente.

Sicurezza e privacy nell'analisi dei requisiti

GDPR

Sostituisce la Data Protection Directive dal 25 maggio 2018. Si deve aderire se si tratta di un prodotto software che tratti i dati personali. Privacy by design e by default, minimalità, proporzionalità, anonimizzazione, pseudonimizzazione, trasferimento dati fuori dall'UE (attenzione al cloud), adeguatezza delle misure di sicurezza. È UN VINCOLO CONSIDERATO FIN DALL'INIZIO.

Pseudonimizzazione

Si trattano i dati personali così che essi non possano essere attribuiti a un interessato specifico, le informazioni aggiuntive vanno conservate separatamente con misure per non associarle.

Principi

I dati personali devono essere trattati in modo LECITO, EQUO e TRASPARENTE nei confronti dell'interessato, raccolti per finalità determinate, esplicite e legittime, in modo non incompatibile con tali finalità. Devono essere adeguati, pertinenti e limitati a quanto necessario rispetto alle finalità (MINIMIZZAZIONE), esatti e aggiornati, con tutte le precauzioni per togliere o aggiornare i dati inesatti. Devono essere conservati in una forma che consenta l'identificazione degli interessati per un arco di tempo non superiore al conseguimento delle finalità per cui sono trattati.

Articolo 25

Il responsabile mette in atto misure tecniche e organizzative adeguate sia al momento di determinare i mezzi del trattamento che all'atto del trattamento, ad esempio la pseudonimizzazione, la minimizzazione a tutela dell'interessato. Il responsabile mette in atto misure che di default trattano solo i dati necessari per le finalità del trattamento e non devono essere accessibili a un numero indefinito di persone fisiche senza l'intervento di una persona fisica.

Articolo 32

Il responsabile e l'incaricato del trattamento mettono in atto misure che garantiscono un livello di sicurezza adeguato: pseudonimizzazione, cifratura dei dati, continua riservatezza (integrità, disponibilità e resilienza dei servizi), la capacità di ripristinare il servizio in caso di guasto, verificare l'efficacia delle misure. Bisogna considerare i rischi presentati dalla distruzione o leaking dei dati. Bisogna aderire a un codice di condotta concorde con l'articolo 40 o un meccanismo di certificazione conforme all'articolo 42.

Trasferimento dei dati a paesi terzi

È ammesso se la Commissione ha deciso che il paese terzo garantisce un livello di protezione adeguato, non servono autorizzazioni specifiche. Oppure se non c'è stata la decisione si può fare solo se ci sono garanzie adeguate e che ci siano diritti azionabili dagli interessati con mezzi di ricorso effettivi per gli interessati.

Sicurezza informatica

Salvaguardia dei sistemi informatici da potenziali rischi o violazioni dati. L'obiettivo dell'attacco è il contenuto informativo. La sicurezza informatica impedisce l'accesso ai non autorizzati, regola l'accesso ai diversi soggetti, si evita la copia di dati del sistema informatico.

Cosa proteggere

L'informazione, la sua riservatezza, integrità e autenticità, disponibilità, con un accesso controllato (IDENTIFICAZIONE, AUTENTICAZIONE, AUTORIZZAZIONE), l'affidabilità del funzionamento.

Violazioni

Tentativi non autorizzati di accesso a zone riservate, furto di identità digitale, uso di risorse non di propria competenza, attacchi DoS che rendono inusabili alcune risorse per danneggiare gli utenti.

Fattori influenti

Quando si scelgono le misure di sicurezza incidono diverse caratteristiche: dinamicità. Dimensione e tipo di accesso, tempo di vita, costo di generazione, costo in caso di violazione, valore percepito e tipologia di attaccante. La forza di un sistema è data dalla forza dell'anello più debole che lo compone.

Protezione fisica

Che non sia la più facilmente attaccabile. Vanno bene i criteri preesistenti la sicurezza informatica e che si conosca il comportamento dei sistemi (percorsi accessibili, copie temporanee). Un'autenticazione forte è data da qualcosa che si conosce (CONOSCENZA), possiede (POSSESSO) ed è (CONFORMITÀ).

Crittografia

La simmetrica garantisce riservatezza, non identifica né autentica. L'asimmetrica ha come obiettivo identificare quindi può anche autenticare e rappresentare la paternità. Si fa ricorso a terze parti per certificare l'autenticità di una chiave pubblica. La cifratura simmetrica non garantisce autenticità, il numero di chiavi per utente è grande, non protegge. La cifratura asimmetrica non è efficiente, non è robusta, la lunghezza delle chiavi è grande.

Crittografia simmetrica

È implementata con cifrari segreti, con tecniche derivanti dalla teoria dell'informazione (CONFUSIONE e DIFFUSIONE), una singola chiave cifra e decifra.

Crittografia asimmetrica

È moderna (1976), basata sulla teoria della complessità computazionale. Abbiamo 2 chiavi non facilmente calcolabili: PRIVATA, identifica il possessore; PUBBLICA, verifica l'uso di una chiave privata. Si cifra con la pubblica e si decifra con la chiave privata.

Firma digitale

Si calcola l'hash e si cifra con la chiave privata. Si invia il documento inalterato con la firma, si confronta l'hash del documento con l'hash decifrato con la chiave pubblica.

Biometria

È un cardine per l'autenticazione forte: qualcosa che sei, hai o sai. È ostinatamente usata per l'identificazione, nonostante le lunghe operazioni di confronto e un cattivo bilanciamento con falsi positivi e falsi negativi, peggiori prestazioni. Va meglio per l'autenticazione, con un solo confronto. Però se si compromettono i dati biometrici non è possibile la sostituzione.

Perché un sistema sicuro

Dipende da vincoli tecnici, amministrativi, politici. Bisogna definire un piano di sicurezza. La sicurezza è un processo complesso, una serie di caratteristiche (a livello di rete, a livello di applicazione, protezione dei dati sensibili). La sfida è sviluppare applicazioni che tengano conto di questi aspetti dall'inizio.

Sistemi critici

Sono sistemi tecnici o socio-tecnici da cui dipendono persone o aziende. Se non forniscono servizi si possono verificare perdite importanti. Esistono i sistemi SAFETY CRITICAL, in cui i fallimenti possono provocare incidenti, MISSION CRITICAL, i malfunzionamenti possono causare fallimenti di attività, BUSINESS CRITICAL, in cui i fallimenti possono portare a costi alti per le aziende. È importantissima per un sistema critico la fidatezza, cioè deve essere un sistema DISPONIBILE, AFFIDABILE, SICURO e PROTETTO. I fallimenti possono essere causati dall'hardware (guasti, errori di progettazione, componenti esauriti), software (errori nelle specifiche, progettazione o implementazione), operatori umani (errori), con il tempo questi ultimi sono la minaccia più grave. Con Internet è diventata sempre più importante la protezione dei sistemi critici. Le connessioni di rete espongono il sistema ad attacchi, però la rete permette di diffondere velocemente i dettagli sulle vulnerabilità. Si può attaccare con virus, usi non autorizzati di servizi, modifiche non autorizzate al sistema e ai suoi dati.

Esempi di attacchi

EXPLOIT, si sfrutta un bug per l'acquisizione di privilegi. BUFFER OVERFLOW, scrivere più dati del previsto in modo da sovrascrivere zone di memoria con dati o stack. SHELL CODE, sequenza di caratteri che lanciano una shell per acquisire accesso alla linea di comando. SNIFFING, intercettazione passiva di dati che transitano in una rete. CRACKING, modifica di un software per rimuovere la protezione dalla copia o accesso ad un'area riservata. SPOOFING, si simula un IP privato da una rete pubblica facendo credere agli host che l'IP della macchina server sia il suo. TROJAN, programma con funzionalità maliziose. DENIAL OF SERVICE, il sistema è forzato a una condizione che impedisce l'esecuzione dei servizi, influenzando la disponibilità del sistema.

Ingegneria della sicurezza

Fa parte della sicurezza informatica. Si occupa di come costruire sistemi resistenti a minacce. Bisogna considerare la sicurezza dell'applicazione e dell'infrastruttura in cui il sistema è costruito.

Applicazione e infrastruttura

La sicurezza dell'applicazione è garantita dal fatto che il sistema sia progettato per resistere agli attacchi, mentre la sicurezza dell'infrastruttura è un problema manageriale che deve essere affrontato in modo da configurare l'infrastruttura per resistere agli attacchi. La struttura deve essere inizializzata in modo che tutti i servizi di sicurezza siano disponibili e bisogna monitorare e riparare eventuali falle di sicurezza che emergono durante l'uso del software.

Gestione della sicurezza

La gestione degli utenti e dei permessi riguarda l'inserimento e la rimozione di utenti dal sistema, l'autenticazione e la creazione di appropriati permessi. Il deployment e il mantenimento del sistema riguarda l'installazione e configurazione dei software e middleware, l'aggiornamento periodico del software con tutte le patch disponibili. Il controllo degli attacchi, la rilevazione e il ripristino riguardano il controllo del sistema per accessi non autorizzati, l'identificazione e messa in opera di strategie contro gli attacchi, il backup per ripristinare il normale utilizzo dopo un attacco.

Glossario della sicurezza

BENE (ASSET), risorsa del sistema da proteggere; ESPOSIZIONE (EXPOSURE), possibile perdita o danneggiamento come risultato di un attacco riuscito (dati o tempo); VULNERABITÀ (VULNERABILITY), debolezza nel sistema software sfruttabile per causare una perdita o un danno; ATTACCO (ATTACK), sfruttamento di una vulnerabilità; MINACCIA (THREAT), circostanza che può causare perdite e danni; CONTROLLO (CONTROL), misura protettiva che riduce le vulnerabilità.

Tipi di minacce

ALLA RISERVATEZZA, informazioni svelate a soggetti non autorizzati; ALL'INTEGRITÀ, si possono danneggiare o corrompere i dati o il software; ALLA DISPONIBILITÀ, può essere negato l'accesso a utenti autorizzati. Sono minacce interdipendenti.

Tipi di controllo

PER GARANTIRE CHE GLI ATTACCHI NON ABBIANO SUCCESSO, si protegge il sistema disconnettendolo o crittografando, si evitano problemi di sicurezza; PER IDENTIFICARE E RESPINGERE GLI ATTACCHI, si monitorano le operazioni di sistema e si monitorano pattern di attività atipici; PER IL RIPRISTINO, backup, replicazione, assicurazioni.

Analisi del rischio

Si occupa di valutare le possibili perdite che un attacco può causare ai beni di un sistema e bilancia le perdite con i costi richiesti per la protezione dei beni stessi. Il costo di protezione è molto minore del costo della perdita. È una problematica più manageriale che tecnica, quindi gli ingegneri della sicurezza forniscono una guida tecnica e giuridica sui problemi di sicurezza. Il manager dovrà accettare i costi di sicurezza o i rischi. L'analisi del rischio inizia valutando le politiche di sicurezza organizzazionali, cosa si può fare e cosa non. Le politiche di sicurezza propongono le condizioni da mantenere nel sistema di sicurezza. È un processo in più fasi: valutazione preliminare del rischio, ciclo di vita della valutazione del rischio (con quello dello sviluppo software).

Analisi del sistema informatico

Si può stabilire l'agenda delle attività: analisi risorse fisiche, logiche e delle dipendenze fra risorse.

Analisi delle risorse fisiche

Il SI viene visto come insiemi di dispositivi che, per funzionare, hanno bisogno di spazio, alimentazione, ambiente, protezioni da furi o danni. Bisogna individuare tutte le risorse fisiche, ispezionare tutti i locali che ospitano le risorse fisiche, verificare la cablatura dei locali.

Analisi delle risorse logiche

Il SI viene visto come insieme delle informazioni, flussi e processi. Occorre classificare le informazioni in base al valore, il grado di riservatezza e il contesto di afferenza. E poi anche classificare i servizi offerti, affinché non presentino effetti collaterali pericolosi per la sicurezza del sistema.

Analisi delle dipendenze tra risorse

Per ciascuna risorsa occorre individuare di quali altre risorse essa ha bisogno per funzionare correttamente. Questa analisi tende a evidenziare le risorse potenzialmente critiche, da cui dipende il funzionamento di altre risorse. I risultati di questa analisi sono usati anche nella fase di valutazione del rischio, sono di supporto allo studio della propagazione di malfunzionamenti dopo eventi indesiderati.

Identificazione delle minacce

Si cerca di definire cosa non deve poter accadere al sistema. Si parte dal considerare come evento indesiderato qualunque accesso non esplicitamente permesso. È possibile distinguere tra attacchi intenzionali ed eventi accidentali.

Attacchi intenzionali

Vengono caratterizzati in funzione della risorsa attaccate e dalle tecniche usate. Queste si classificano in funzione del livello su cui operano: LIVELLO FISICO, furto (attacco a disponibilità e riservatezza), danneggiamento (attacco alla disponibilità e integrità); LIVELLO LOGICO, sottrarre informazione o degradare l'attività del sistema, intercettazione o deduzione (attacco alla riservatezza, sniffing, spoofing...), intrusione (attacco all'integrità e alla riservatezza, IP-spoofing, backdoor...), disturbo (attacco alla disponibilità, virus, worm, denial of service...).

Eventi accidentali

Anch'essi si verificano a livello fisico (guasti a dispositivi di sistema o di supporto) e logico (perdita password o chiave hardware, cancellazione di file, corruzione del software di sistema ad esempio a seguito di installazione di estensioni incompatibili).

Valutazione dell'esposizione

Associamo un rischio a ogni minaccia, così indirizziamo l'attività di individuazione di contromisure. Rischio vuol dire una combinazione di probabilità che un evento accada con il danno che può arrecare. Si tiene conto delle dipendenze tra risorse e della propagazione del malfunzionamento.

Valutazione delle probabilità: attacchi intenzionali

La probabilità dipende da quanto è facile attuare l'attacco e i vantaggi che se ne può trarre. Il danno si misura nel grado di perdita dei tre requisiti fondamentali (RISERVATEZZA, INTEGRITÀ, DISPONIBILITÀ). Bisogna anche valutare un attacco composto, cioè attacchi su tutte le risorse attaccabili dagli strumenti dell'attaccante, con lo stesso obiettivo, in sequenza.

Individuazione del controllo

Bisogna scegliere il controllo da adottare. Si valuta il rapporto costo efficacia, si analizzano gli standard e i modelli di riferimento. Si fa il controllo di carattere organizzativo e il controllo di carattere tecnico.

Valutazione del rapporto costo/efficacia

Valuta il grado di adeguatezza di un controllo, vuole evitare che i controlli presentino un costo ingiustificato rispetto al rischio protetto. L'efficacia del controllo è definita come funzione del rischio rispetto agli eventi indesiderati che neutralizza. Il costo di un controllo deve essere calcolato senza dimenticare i costi nascosti.

Costi nascosti

Ci sono limitazioni imposte dai controlli e dalle operazioni introdotte nel workflow del sistema e dell'organizzazione. Le principali voci di costo sono: il costo di messa in opera del controllo; peggioramento dell'ergonomia dell'interfaccia utente; decadimento delle prestazioni; aumento della burocrazia.

Controlli di carattere organizzativo

La tecnologia funziona bene solo se usata in modo corretto e consapevole dall'utente. Bisogna definire precisamente ruoli e responsabilità. Per ogni ruolo devono essere definite norme comportamentali e procedure precise.

Controlli di carattere tecnico

DI BASE, a livello di SO e servizi di rete. SPECIFICI DEL SISTEMA, si trovano nel livello applicativo. TECNICI PIÙ FREQUENTI, configurazione sicura del SO di server e postazioni di lavoro (contromisura di base), confinamento logico delle applicazioni server su server dedicati. ETICHETTATURA DELLE INFORMAZIONI, con scopo di avere un controllo più fine dei diritti di accesso; MODULI SOFTWARE DI CIFRATURA, integrati con le applicazioni; APPARECCHIATURE di telecomunicazione in gradi di cifrare il traffico in modo trasparente alle applicazioni; FIREWALL E SERVER PROXY in corrispondenza di collegamenti con reti TCP/IP; CHIAVI HARDWARE O DISPOSITIVI DI RICONOSCIMENTO degli utenti basati su rilevamenti biofisici.

Integrazione dei controlli

Un insieme di controlli deve essere ben integrato, non controlli non correlati. La politica di sicurezza deve essere organica. Bisogna selezionare i controlli adottando un sottoinsieme di costo minimo che rispetti vincoli: COMPLETEZZA DELLE CONTROMISURE, OMOGENETITÀ DELLE CONTROMISURE, RIDONDANZA CONTROLLATA DELLE CONTROMISURE, EFFETTIVA ATTUABILITÀ DELLE CONTROMISURE.

Vincoli del sottoinsieme

COMPLETEZZA, deve fare fronte a eventi indesiderati. OMOGENEITÀ, contromisure compatibili e integrabili tra loro. RIDONDANZA CONTROLLATA, dato che ha un costo e deve essere rivelata e vagliata accuratamente. EFFETTIVA ATTUABILITÀ, l'insieme delle contromisure deve rispettare i vincoli imposti dall'organizzazione nella quale andrà a operare.

Ciclo di vita della valutazione del rischio

Serve conoscere l'architettura del sistema e l'organizzazione dei dati. La piattaforma, il middleware e la strategia di sviluppo sono già stati scelti. Si hanno molti più dettagli su cosa proteggere. Le vulnerabilità possono essere ereditate da scelte di progettazione. La valutazione del rischio dovrebbe essere parte di tutto il ciclo di vita del software, dall'ingegnerizzazione al deployment. Il processo è simile a quello della valutazione dei rischi, aggiungendo la valutazione delle vulnerabilità. Essa identifica beni che hanno più probabilità di essere colpiti dalle vulnerabilità. Vengono messe in relazione vulnerabilità con attacchi al sistema. Il risultato della valutazione del rischio è un insieme di decisioni ingegneristiche che influenzano la progettazione o implementazione del sistema o limitano il modo in cui esso è usato.

Security use case e misuse case

I misuse case si concentrano sulle interazioni tra applicazione e attaccanti che cercano di violarla. La condizione di successo è l'attacco andato a buon fine. Sono adatti per analizzare le minacce, ma non sono utili per determinare i requisiti di sicurezza. I security use case specificano i requisiti.

Linee guida per i security use case

Non devono mai specificare i meccanismi di sicurezza, vanno lasciati alla progettazione. I requisiti sono differenziati dalle informazioni secondarie (interazioni del sistema, azioni del sistema e postcondizioni). Bisogna evitare vincoli progettuali non necessari. Poi occorre documentare i percorsi individuali nei casi d'uso per specificare i requisiti di sicurezza. Basarli su differenti tipi di sicurezza fornisce una naturale organizzazione dei casi d'uso. Bisogna documentare le minacce di sicurezza che giustificano i percorsi di sicurezza attraverso i casi d'uso. Distinguere tra interazioni degli utenti e attaccanti. Distinguere tra interazioni visibili esternamente e nascoste del sistema. Documentare si precondizioni che post condizioni che catturano l'essenza dei percorsi individuali.

Requisiti di sicurezza

Non si possono sempre quantificare. Quindi bisogna esprimerli nella forma del non dover. Si definiscono i comportamenti inaccettabili del sistema. Non definiscono funzionalità del sistema. Di solito nella specifica ci si basa sul contesto, sui beni da proteggere e del loro valore per l'organizzazione. Si dividono in requisiti di: IDENTIFICAZIONE, specificano se un sistema deve identificare gli utenti prima di interagire con loro; AUTENTICAZIONE, come identificare gli utenti; AUTORIZZAZIONE, i privilegi d'accesso degli utenti; IMMUNITÀ, come il sistema si protegge da virus; INTEGRITÀ, come evitare la corruzione dei dati; SCOPERTA DI INTRUSIONI, quali meccanismi usare per scoprire gli attacchi; NON RIPUDIAZIONE, una parte interessata in una transazione non può negare il proprio coinvolgimento; RISERVATEZZA, come mantenere la riservatezza delle informazioni; CONTROLLO DELLA PROTEZIONE, come può essere controllato e verificato l'uso del sistema; PROTEZIONE DELLA MANUTENZIONE DEL SISTEMA, come un'applicazione può evitare modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione.

Diagrammi UML

Unified Modeling Language

È un linguaggio per specificare, costruire, documentare un sistema e gli elaborati prodotti durante il suo sviluppo. Ha una notazione e semantica standard basate su un METAMODELLO che definisce i costrutti forniti dal linguaggio, sono ESTENSIBILI o PERSONALIZZABILI. Avvolge tutto il ciclo di vita del software. Combina E/R, workflow, modellazione a oggetti e a componente. Ha diagrammi standard che mostrano tante viste architetturali del modello di sistema. NON È UN MODELLO DI SVILUPPO, è un linguaggio. Propone un ricco insieme di elementi a livello utente. È informale per come usarli.

Diagrammi

Diagrammi di STRUTTURA: classi, strutture composite, componenti, deployment, package, profili. Diagrammi di COMPORTAMENTO: casi d'uso, stato, attività, INTERAZIONE (comunicazione, tempi, sintesi delle interazioni, sequenza).

Package

È usato per raggruppare elementi e fornire loro un NAMESPACE, cioè una porzione del modello in cui possono essere definiti e usati nomi. Un package può essere innestato in altri package e in un namespace ogni nome ha un significato univoco.

Diagramma dei package

È un diagramma che illustra come gli elementi di modellazione sono organizzati in package e le dipendenze (relazioni) tra essi. Quando si usa il diagramma dei package per definire la parte strutturale dell'architettura logica bisogna ricordare che si stanno esprimendo DIPENDENZE LOGICHE che sussistono tra entità del problema. Non è detto che tali dipendenze rimangano in progettazione.

Dipendenze

Possiamo rappresentare relazioni che non sussistono tra istanze del dominio, ma tra gli elementi del modello UML stesso o fra astrazioni. Una dipendenza è rappresentata da una LINEA TRATTEGGIATA ORIENTATA CHE VA DALL'ELEMENTO DIPENDENTE A QUELLO INDIPENDENTE. I cambiamenti dell'indipendente influenzano il dipendente, ne modificano il significato. Il tipo di dipendenza più usato è USE.

Interfaccia

Fornisce un modo per partizionare e caratterizzare gruppi di proprietà. Non deve specificare come va implementata, ma solo ciò che serve per realizzarla. Le entità che la realizzano forniranno una vista pubblica conforme all'interfaccia. Se un'interfaccia dichiara un attributo, non è detto che l'implementazione lo abbia, apparirà così a un osservatore esterno. La notazione è un cerchio vuoto come interfaccia fornita, una mezza luna per un'interfaccia richiesta, una freccia chiusa vuota tratteggiata verso l'interfaccia per l'implementazione.

Diagramma delle classi

Descrive il tipo degli oggetti di un sistema ed i tipi di relazioni statiche tra loro. Mostrano proprietà e operazioni di una classe, con i vincoli applicati alla classe e alle relazioni tra classi. Le proprietà rappresentano le caratteristiche strutturali di una classe (un unico concetto, con notazioni diverse, cioè attributi e associazioni, simili).

Classe

Modella un insieme di entità con lo stesso tipo di caratteristiche (attributi, associazioni, operazioni). Ogni classe ha un nome e un insieme di feature.

Attributi

Ogni attributo descrive una proprietà con una riga di testo all'interno del box della classe. Si esprime nella forma VISIBILTÀ NOME : TIPO MOLTEPLICITÀ=DEFAULT {STRINGA DI PROPRIETÀ} (stringa: String[10] = "Pippo" {readOnly}). È necessario solo il nome.

Molteplicità

Quanti oggetti possono far parte di una proprietà. Si indicano con gli estremi inferiore e superiore (1, 0..1, *). Il valore di un attributo multivalore si indica mediante un insieme.

Visibilità

Può essere + PUBLIC, - PRIVATE, ~ PACKAGE, # PROTECTED.

Operazioni

Sono azioni che la classe sa eseguire, si fanno corrispondere ai metodi. La sintassi è VISIBILITÀ NOME (LISTA PARAMETRI) : TIPO RITORNO {STRINGA DI PROPRIETÀ}.

Operazioni e attributi statici

Sono static operazioni e attributi che si applicano alla classe e non all'istanza. Vanno sottolineati nel diagramma.

Associazioni

Sono un altro modo per rappresentare le proprietà. È una linea continua che collega due classi, orientata dalla sorgente alla destinazione. Si indicano nome e molteplicità agli estremi. Assegnare dei nomi consente la leggibilità del diagramma. L'associazione bidirezionale è formata da una coppia di proprietà collegate, si possono navigare in entrambi i versi, dove sono anche poste le frecce di navigabilità. Con associazioni ternarie o più si introduce il diamante. È il numero minimo e massimo di istanze dell'associazione a cui un'istanza dell'entità può partecipare. Le classi di un'associazione aggiungono caratteristiche proprie dell'applicazione, ma ci può essere solo un'istanza di una coppia di oggetti associati.

Aggregazione e composizione

L'aggregazione è un'associazione che indica la relazione intero-parte. È un diamante vuoto, posto vicino agli interi. È una relazione binaria e può essere ricorsiva. La composizione è un'aggregazione in cui le parti possono essere incluse al massimo un intero ogni istante e solo l'oggetto intero può creare e distruggere le sue parti. È un diamante pieno vicino agli interi. Se l'oggetto che compone viene distrutto anche i figli vengono distrutti, anche se creati o distrutti in momenti diversi dalla creazione o distruzione dell'intero, può essere ricorsiva.

Generalizzazione

È indicata con una freccia vuota orientata verso la superclasse. Ogni istanza dalla sottoclasse è anche una superclasse. Una superclasse può far parte di più generalizzazioni, disgiunte o no, complete o no. Una sottoclasse può anche sovrascrivere (ridefinire) le proprietà ereditate.

Generalizzazione multipla

La generalizzazione singola indica che un oggetto appartiene a un solo tipo, che può ereditare dai tipi del padre. Se multipla, un oggetto può essere descritto da più tipi anche non collegati da ereditarietà. Non è l'ereditarietà multipla, perché in questo caso l'oggetto può essere associato a più tipi senza doverne specificare un altro. Bisogna rendere le combinazioni legali, UML mette ogni generalizzazione in un insieme di generalizzazione. La freccia va etichettata con il nome del rispettivo insieme. La singola è un singolo anonimo insieme di generalizzazione. Gli insiemi di generalizzazione sono disgiunti per definizione.

Sintassi delle relazioni tra classi

Linee tratteggiate: DIPENDENZA (freccia aperta), IMPLEMENTAZIONE DI INTERFACCIA (freccia vuota). Linee continue: GENERALIZZAZIONE (freccia aperta), ASSOCIAZIONE (nessuna freccia), AGGREGAZIONE (rombo vuoto), COMPOSIZIONE (rombo pieno).

Classi astratte

Sono classi non direttamente istanziabili, bisogna creare una sottoclasse concreta. Ha una o più operazioni astratte. Si indica l'operazione o la classe astratta con il nome in corsivo. Si possono anche indicare proprietà astratte, anche astraendone i metodi d'accesso. Fanno da superclassi comuni. Le sottoclassi sono in qualche modo intercambiabili e sostituibili con la superclasse.

Enumerazioni

Sono usate per mostrare un insieme di valori prefissati che non hanno altre proprietà se non il loro valore simbolico. Si rappresentano con una classe marcata <<enumeration>>.

Diagramma di sequenza

Illustra le interazioni tra classi ed entità disponendole in una sequenza temporale. Mostra i soggetto (LIFELINE) che partecipano all'interazione e la sequenza di messaggi scambiati. In ascissa abbiamo i soggetti (non per forza in ordine di esecuzione) e in ordinata la scala dei tempi verso il basso.

Lifeline

La vita dei partecipanti a un diagramma di sequenza è rappresentata da una lifeline, cioè una linea tratteggiata verticale etichettata. In caso di un partecipante entità composta si possono fare lifeline interne. L'ordine delle OccurrenceSpecification (scambio eventi) è quello in cui si deve verificare. La distanza grafica tra eventi non ha rilevanza. Il rettangolino posto sulla lifeline è l'attivazione del metodo.

Vincoli temporali

Per sistemi real time è necessario specificare un istante in cui il messaggio deve essere mandato, Time Constraint e Duration Constraint vengono in aiuto.

Riferimento ad altri diagrammi

Possono essere complessi i diagrammi, per cui si possono inserire riferimenti ad altri diagrammi e passare gli argomenti (ovviamente solo se li accetta). I riferimenti si chiamano InteractionUse, i punti di connessione di chiamano Gate (relazione tra messaggio fuori dal frammento e all'interno del frammento).

Messaggio

È un'interazione tra lifeline, si può creare o distruggere un'istanza, invocare un'operazione, emettere un segnale. Il COMPLEETE MESSAGE è il messaggio che specifica mittente e destinatario, solo con il mittente è un LOST MESSAGE (noto evento di invio), solo con il destinatario è un FOUND MESSAGE (noto evento di ricezione), con nessuno dei due è un UNKNOWN MESSAGE. La riga continua freccia piena indica un MESSAGGIO SINCRONO; la riga continua freccia vuota indica un MESSAGGIO ASINCRONO; la riga tratteggiata freccia vuota indica il RITORNO DEL MESSAGGIO.

CombinedFragment

Sono contenitori che delimitano un'area di interesse del programma, spiegano che una serie di eventi accadranno in base alla semantica associata, hanno un operatore ciascuno e una guardia. Il LOOP indica che ciò che è racchiuso verrà eseguito finché la guardia sarà verificata. Le ALTERNATIVES indicano che sarà eseguito il contenuto di uno solo degli operandi, dove la guardia è verificata. L'OPTIONAL indica che l'esecuzione avviene se la guardia è verificata. Il BREAK indica che l'interazione sarà terminata. Il CRITICAL indica un blocco di esecuzione atomico. Il PARALLEL indica che il contenuto del primo operando può essere eseguito in parallelo a quello del secondo. Il WEAK SEQUENCING indica che il risultato può essere una combinazione qualsiasi delle interazioni ottenute, purché si mantenga l'ordinamento degli operandi e gli eventi con gli stessi destinatari seguano l'ordine. Lo STRICT FREQUENCING indicano che il contenuto deve essere eseguito anche nell'ordine degli operandi. L'IGNORE indica che alcuni messaggi importanti per il sistema non sono rappresentati perché non servono per capire l'interazione. Il CONSIDER è complementare ad IGNORE. Il NEGATIVE racchiude una sequenza che non si deve mai verificare. L'ASSERTION racchiude l'unica sequenza valida, si associa all'utilizzo di uno State Invariant come rinforzo.

Diagramma di stato

Modellano la dipendenza tra classe/entità e messaggi/eventi ricevuti in ingresso. Specifica il ciclo di vita di una classe, definendo le regole che lo governano. Una classe in uno stato può essere interessata a eventi. La classe può passare a un nuovo stato (transizione). Uno stato è una condizione nella vita di un oggetto in cui soddisfa una condizione, esegue un'attività o esegue un evento. L'evento è un'occorrenza nel tempo e nello spazio. Gli stati vengono indicati con rettangoli arrotondati, le transizioni con frecce, gli eventi scrivendo il nome del messaggio con argomenti, i marker di inizio e fine con un cerchio nero con una freccia che punta allo stato iniziale e con un cerchio nero racchiuso da un anello sottile. Il vertice è l'astrazione di nodo nel diagramma e può essere sorgente o destinazione di una o più transizioni. Le azioni sono eseguibili dall'entità in risposta a un evento.

Stato

Modella una situazione statica, come dinamiche, ad esempio se modella un comportamento (sta nello stato entra nello stato quando il comportamento inizia e esce quando il comportamento è completato). Il COMPOSITE STATE può contenere una regione o è decomposto in più regioni ortogonali. Ogni regione ha sottovertici mutuamente esclusivi e il proprio insieme di transizioni, ogni stato di una regione è un substato, la transizione verso il marker finale della regione è il completamento del comportamento della regione e una volta che tutte le regioni ortogonali hanno completato il loro comportamento, è completato il comportamento dello stato. Il SIMPLE STATE è uno stato senza sottostati. Il SUBMACHINE STATE specifica l'inserimento di una sottoparte e fattorizza comportamenti comuni, è equivalente al composite state, le azioni sono definite come parte dello stato. Uno stato può essere ridefinito.

Pseudostato

Vengono usati per collegare più transizioni in percorsi più complessi. L'INITIAL è la sorgente della singola transizione verso lo stato di default di un composite state. DEEPHISTORY è la più recente configurazione attiva del composite state che contiene direttamente questo pseudostato. SHALLOWHISTORY è il più recente substate attivo di un composite state. JOIN permette di eseguire il merge di tante transizioni. Il FORK separa una transizione entrante in più transizioni che vanno in regioni ortogonali. Il JUNCTION è un vertice privo di semantica che si usa per incatenare transizioni multiple, si usa per creare percorsi di transizione composti tra stati, si possono far convergere transizioni multiple o il contrario, un branch condizionale statico, la guardia di default else viene attivata quando tutte le guardie sono false. La CHOICE quando viene raggiunta causa la valutazione dinamica delle guardie dei trigger delle transizioni uscenti, le guardie sono come funzioni valutate al momento del raggiungimento del vertice, realizza un branch condizionale dinamico separando transizioni uscenti e creando diversi percorsi di uscita, con più guardie verificate si sceglie un percorso arbitrario, se nessuna è verificata si chiama ill formed, è consigliato definire il percorso else. L'ENTRY POINT è l'ingresso di uno state machine o un composite state. L'EXIT POINT è l'uscita. Il TERMINATE indica che l'esecuzione della state machine è conclusa.

Transizione

È una relazione diretta tra vertice di origine e destinazione. Può essere parte di una transizione composta che porta la macchina a stati da una configurazione di stato all'altra. È possibile vedere che tra proprietà di una transizione compaiono diversi concetti rilevanti come il TRIGGER (innesco di transizione), la GUARDIA (controllo fine sull'innesco della transizione), l'EFFETTO (comportamento opzionale, l'azione, da eseguire quando scatta la transizione).

Tipi di eventi

DI CHIAMATA, ricezione di un messaggio che esegue l'operazione. DI CAMBIAMENTO, si verifica quando una condizione passa da falsa a vera (si usa <<when>> seguito da un'espressione, utile per descrivere la situazione in cui un oggetto cambia stato perché il valore dei suoi attributi è modificato dalla risposta a un messaggio inviato). SEGNALE, la ricezione di un segnale. TEMPORALE, denota un lasso di tempo che deve trascorrere dopo un evento dotato di nome (<<after>> può far riferimento all'istante in cui l'oggetto è entrato nello stato corrente, <<at>> esprime qualcosa che deve accadere in un particolare momento).

Azione

ENTRY, quando si entra in uno stato si genera un evento di entrata associato a comportamenti eseguiti prima che qualsiasi altra azione venga eseguita. EXIT, il viceversa, prima che lo stato venga lasciato. DO, il comportamento eseguito nello stato, dopo l'entry, quando si conclude invoca l'exit se specificata la transizione, ma se c'è una transizione d'uscita prima che finisca abortisce e l'exit si esegue. L'entry c'è per fattorizzare comportamenti comuni all'ingresso di uno stato, come le exit, non c'è però la possibilità di esprimere condizioni di guardia sui comportamenti.

Diagramma di stato

Il diagramma deve essere il più semplice possibile, per evitare metodi con molti blocchi condizionali, un arduo testing, difficoltà date dalla complessità di un comportamento che dipende molto dallo stato. Se una classe ha molti stati si può considerare di scomporre la classe in due classi diverse, non è una regola universale e può essere rischioso.

Diagramma delle attività

Descrivono il modo in cui diverse attività sono coordinate e possono essere usati per mostrare come implementare un'operazione. Mostrano le attività di un sistema generale e delle sottoparti, soprattutto se ha tanti obiettivi e si vogliono modellare le dipendenze prima di decidere in che ordine svolgere queste azioni. Sono utili anche per descrivere i casi d'uso e le loro eventuali dipendenze. Modellano un processo, organizzano più entità in un insieme di azioni secondo un flusso. L'ATTIVITÀ è un lavoro atomico che deve essere svolto, possono uscire più archi. L'ARCO è una freccia, come una transizione, però non può essere etichettato con eventi o azioni, ma con un'azione di guardia se l'attività successiva lo chiede. L'OGGETTO rappresenta un oggetto importante usato come I/O di azioni. La SOTTOATTIVITÀ nasconde un diagramma delle attività interno a un'attività. START e END POINT sono i punti di inizio e fine di un diagramma, gli end point possono non esserci o essercene più di uno. Il CONTROLLO è un nodo che descrive il flusso delle attività. Delle entità token viaggiano lungo il diagramma, il loro flusso è quello delle attività, possono rimanere fermi in un nodo azione/oggetto in attesa che si avveri una condizione sull'arco o una pre o postcondizione su un nodo, il suo movimento è atomico, un nodo azione viene eseguito se sono presenti token su tutti gli archi in entrata e le precondizioni sono soddisfatte e al termine sono generati token su tutti gli archi di uscita. DECISION e MERGE determinano il comportamento condizionale: decision ha una transizione entrante e più transizioni uscenti in cui solo una è prescelta; il merge ha più transizioni entranti e una uscente che conclude il blocco condizione cominciato con un decision. FORK e JOIN determinano il comportamento parallelo: alla transizione entrante scattano le transazioni che escono dal fork senza una sequenza particolare, il fork può avere guardie e per la sincronizzazione c'è il join.

Analisi del problema

Introduzione

L'obiettivo è esprimere fatti oggettivi sul problema focalizzandosi su sottosistemi, ruoli e responsabilità degli scenari durante l'analisi dei requisiti senza descrivere la sua soluzione. Il risultato è l'architettura logica, il piano di lavoro e il piano di collaudo. L'ANALISI DEL DOCUMENTO DEI REQUISITI si concentra sull'analizzare funzionalità e rischi evidenziati nel documento dei requisiti; l'ANALISI DEI RUOLI E DELLE RESPONSABILITÀ analizza i ruoli emersi nei casi d'uso e pone attenzione nell'attribuzione delle responsabilità a tali ruoli tenendo conto dell'analisi del rischio; la SCOMPOSIZIONE DEL PROBLEMA si occupa di suddividere il problema in sottoproblemi se possibile; la CREAZIONE DEL MODELLO DEL DOMINIO costruisce il diagramma delle classi del dominio; ARCHITETTURA LOGICA: STRUTTURA indica la creazione di diagrammi strutturali (package e classi) dell'architettura logica; ARCHITETTURA LOGICA: INTERAZIONE è la fase in cui si creano i diagrammi di interazione (sequenza) dell'architettura logica; ARCHITETTURA LOGICA: COMPORTAMENTO è quando si creano diagrammi di comportamento (stato e attività) dell'architettura logica; la DEFINIZIONE DEL PIANO DI LAVORO assegna le responsabilità ai membri del team; DEFINIZIONE DEL PIANO DI COLLAUDO definisce i risultati attesi da ogni classe o sottosistema dell'architettura logica.

Architettura logica

È un insieme di modelli UML costruiti per definire una struttura concettuale del problema robusta e modificabile. Con l'architettura logica l'analista descrive la struttura, il comportamento atteso, le interazioni. Possono essere dedotte dai requisiti caratteristiche del problema e del dominio senza riferimento a tecnologie realizzative.

Piano di lavoro

Esprime l'articolazione in termini di risorse umane, temporali, di elaborazione necessarie allo sviluppo del prodotto dall'analisi del problema.

Piano di collaudo

Definisce i risultati attesi dalle entità dell'architettura logica da sollecitazioni. Si possono fornire test per impostare il piano di collaudo, facilita il lavoro della successiva fase di progetto.

Analisi documento dei requisiti

Il documento evidenzia le funzionalità e i servizi che dovranno essere sviluppati (REQUISITI FUNZIONALI), i vincoli da considerare (NON FUNZIONALI), il mondo esterno con cui interagire, i rischi legati ad attacchi di protezione, integrità e privacy dei dati (DI SICUREZZA). Da tale documento bisogna applicare un'analisi di tutto quello che è evidenziato.

Analisi funzionalità

Per ogni funzionalità dei casi d'uso si analizzano: il TIPO, cioè l'obiettivo delle funzionalità (la memorizzazione dati, l'interazione con l'esterno...); le INFORMAZIONI COINVOLTE, un'analisi sistematica di tutte le informazioni su cui la funzionalità opera; il FLUSSO DELLE INFORMAZIONI, le informazioni in input (come si validano) e output (valutare dall'esterno le funzionalità). La tabella di funzionalità indica la funzionalità, il tipo, il grado di complessità, i requisiti collegati. La tabella informazioni indica l'informazione, il tipo (composto o semplice), il livello di riservatezza, se di input o output e i vincoli.

Analisi dei vincoli

Bisogna analizzarne il TIPO, a quale categoria appartiene (performance, tempi di risposta, scalabilità...). Etichettiamo il requisito per evidenziare la categoria, alcuni requisiti influenzano aspetti del sistema, quindi vanno indicati con l'impatto. Le FUNZIONALITÀ COINVOLTE indicano le funzionalità influenzate dal requisito. Possiamo predisporre la tabella vincoli con requisito, categorie, impatto e funzionalità coinvolte.

Analisi delle interazioni

Ci sono interazioni con UMANI e con SISTEMI ESTERNI. Per quelle con gli umani si analizzano le interfacce identificate col cliente in fase di analisi dei requisiti o si delineano possibili interfacce, si individuano le maschere di I/O dati, si individuano le sole informazioni necessarie da mostrare in ogni maschera, si crea un legame tra maschere, informazioni e funzionalità (si può creare una tabella da queste). Per quelle coi sistemi esterni bisogna analizzare i morsetti dei sistemi con cui interagire e individuare il protocollo di interazione, si può fare una tabella dei sistemi esterni (sistema, descrizione, protocollo, livello di protezione).

Analisi di ruoli e responsabilità

Ogni attore dei casi d'uso ha RESPONSABILITÀ (cosa deve fare in ogni funzionalità), le INFORMAZIONI A CUI PUÒ ACCEDERE (di ogni funzionalità), le MASCHERE CHE PUÒ VISUALIZZARE, il suo LIVELLO DI RISERVATEZZA (quale livello è necessario per il suo ruolo), la NUMEROSITÀ ATTESA (numero di persone che possono giocare quel ruolo). Da queste informazioni si può predisporre la tabella ruoli. La tabella ruolo-informazione invece indica l'informazione e il tipo di accesso. C'è una tabella ruoli per tutti i ruoli e una tabella ruolo-informazioni per ogni ruolo.

Scomposizione del problema

Dall'analisi delle funzionalità, ogni funzionalità complessa bisogna vedere se è scomponibile e se ci sono legami con le possibili sottofunzionalità. Si possono predisporre delle tabelle (di scomposizione delle funzionalità e delle sottofunzionalità, in quest'ultima si specificano i legami tra le funzionalità).

Creazione modello del dominio

Si parte dal glossario dell'analisi dei requisiti e dalle tabelle di informazioni e flusso e si costruisce il diagramma delle classi, si riuserà nell'architettura logica come modello dei dati. Non tutti i vocaboli elencati diverranno classi, bisogna evitare ridondanze, nomi di eventi, nomi del metalinguaggio (come requisiti e sistema), nomi fuori dell'ambito del sistema, nomi che possono essere attributi. Bisogna individuare oggetti rilevanti per il problema e limitarsi a quelle classi che fanno parte del vocabolario del dominio del problema. Bisogna individuare le relazioni tra classi e per ogni classi attributi e servizi per l'esterno.

Individuazione delle classi

Eliminare nomi che non si riferiscono a attributi primitivi e operazioni o non classi. Scegliere il termine significativo tra i sinonimi. Deve avere un nome familiare all'utente o esperto del dominio, non allo sviluppatore. Gli aggettivi e gli attributi possono indicare oggetti diversi, usi diversi di un oggetto o essere irrilevanti. Le frasi passive e impersonali devono essere rese attive e esplicite perché nascondono entità. Bisogna individuare gli ATTORI con cui il sistema interagisce, i MODELLI e i loro elementi specifici, le COSE TANGIBILI che fanno parte del dominio, i CONTENITORI di altri oggetti, EVENTI o TRANSIZIONI da gestire. Quando dobbiamo includere una classe chiediamoci se il sistema deve interagire con quella classe e quali sono le sue responsabilità nel contesto. Attributi e oggetti devono essere applicabili a tutti gli oggetti della classe, se esistono attributi o operazioni valide solo per alcuni oggetti della classe si parla di ereditarietà.

Individuazione delle relazioni

La maggior parte degli oggetti interagisce con altri in vari modi. In ogni relazione c'è un cliente che dipende da un fornitore, il cliente ha bisogno del fornitore per funzionalità che non può svolgere autonomamente, quindi per far funzionare bene il cliente deve funzionare bene il fornitore. Si può fare una tabella.

Individuazione dell'ereditarietà

Deve rispecchiare una tassonomia presente nel dominio, non quella dell'implementazione e non va usata solo per caratteristiche comuni.

Individuazione delle associazioni

Un'associazione rappresenta una relazione strutturale tra due istanze di classi diverse o della stessa classe. Può rappresentare un contenimento logico (aggregazione) o fisico (composizione) o non rappresentare un reale contenimento. Nell'aggregazione il contenente contiene il contenuto in modo non esclusivo. Nella composizione lo contiene in modo esclusivo, quindi la cancellazione del contenitore implica la cancellazione del contenuto. Le associazioni molti a molti nascondono classi di associazioni del tipo evento da ricordare.

Individuazione collaborazioni

Una classe ne USA un'altra quando ne ha bisogno per svolgere alcune funzioni che non sarebbe in grado di svolgere da sola. Succede se la classe ha bisogno come argomento un'istanza dell'altra, se la classe restituisce il valore dell'altra o se usa un'istanza dell'altra classe. Non è una relazione simmetrica, attenzione a non far dipendere le classi da sé stesse.

Individuazione degli attributi

Ogni attributo modella una proprietà atomica di una classe: un valore singolo o valori strettamente collegati tra loro. Le proprietà non atomiche sono modellate come associazioni. Ogni oggetto a tempo di esecuzione avrà un valore specifico, l'informazione di stato. La base di partenza sono le tabelle di informazione/flusso, il nome dell'attributo deve essere familiare all'utente, non deve essere un nome di un valore, deve iniziare in minuscolo. Bisogna esprimere i vincoli dell'attributo: TIPO, semplice o enumerativo; OPZIONALITÀ; VALORI AMMESSI, dominio, antidominio, univocità; VINCOLI DI CREAZIONE, valore di default, immodificalità del valore; VINCOLI DOVUTI A VALORI DI ALTRI ATTRIBUTI; UNITÀ DI MISURA; VISIBILITÀ, opzionale in fase di analisi; APPARTENENZA ALLA CLASSE, attributi e associazioni di classe. Si può usare UML, Object Constraint Language, formato libero di un commento UML. In caso di booleano il nome dell'attributo può essere un valore di enumerazione, attenzione. Attributi con valore non applicabile o opzionale o multipli possono nascondere l'ereditarietà o una nuova classe. Per attributi calcolabili bisogna specificare l'operazione e mai l'attributo (memorizzarlo è un compromesso progettuale tra tempo di calcolo e memoria usata). Bisogna applicare l'ereditarietà posizionando attributi e associazioni più generali il più in alto possibile nella gerarchia e viceversa.

Individuazione delle operazioni

Il nome deve appartenere al vocabolario standard del dominio del problema, può essere un verbo imperativo o in terza persona, che sia consistente nel sistema. Le operazioni standard sono operazioni che tutti gli oggetti hanno in quanto tali (costruttori, accessori), sono implicite e non compaiono nel diagramma delle classi di analisi. Le altre devono essere determinate, perché sono servizi offerti da altri oggetti e quindi compaiono. Delle classi contenitori mostrare le operazioni dei calcoli sugli oggetti, le selezioni specifiche, le operazioni di tipo da applicare su tutte le parti. Bisogna distribuire le operazioni in modo bilanciato, metterle vicino ai dati a essa necessari, le operazioni più generali vanno più in alto nella gerarchia e viceversa, bisogna descrivere tutti i vincoli (parametri formali con tipo e significato, precondizioni, postcondizioni, invarianti di classe, eccezioni sollevate, eventi di attivazione, applicabilità dell'operazione). La PRECONDIZIONE è un'espressione logica che riguarda le aspettative sullo stato del sistema prima che venga eseguita un'operazione, esplicita che è responsabilità del chiamante controllare la correttezza dei parametri passati. La POSTCONDIZIONE riguarda le aspettative sullo stato del sistema dopo l'esecuzione dell'operazione. L'INVARIANTE DI CLASSE è un vincolo di classe che deve essere sempre verificato sia prima che dopo le operazioni pubbliche della classe, può non essere verificato durante l'esecuzione dell'operazione. L'ECCEZIONE si verifica quando l'invocazione rispetta le precondizioni ma non è in grado di terminare rispettando le postcondizioni.

Architettura logica: struttura

Si compone di due diagrammi UML: DEI PACKAGE, fornisce una visione di alto livello dell'architettura; DELLE CLASSI, fornisce una visione più dettagliata del contenuto dei package. È opportuno organizzare l'architettura logica seguendo il pattern BOUNDARY CONTROL ENTITY.

Boundary Control Entity

È un pattern che suggerisce di basare l'architettura sulla partizione sistematica degli use case in oggetti di tre categorie: INFORMAZIONE, PRESENTAZIONE, CONTROLLO. A ciascuna di queste dimensioni corrisponde un insieme di classi. È stato introdotto in RUP con icone particolari. L'ENTITY è la dimensione relativa alle entità, l'insieme delle classi che includono funzionalità relative alle informazioni che caratterizzano il problema. Il BOUNDARY è la dimensione relativa alle funzionalità che dipendono dall'ambiente esterno, l'insieme delle classi che incapsulano l'interfaccia del sistema verso il mondo esterno. Il CONTROL è la dimensione relativa agli enti che incapsulano il controllo, il collante tra interfacce e entità. Questa distinzione architetturale è un buon punto di partenza. L'analista partiziona i problemi complessi in sottoproblemi, evita di associare alle entità di un dominio funzionalità specifiche di applicazione e funzionalità tipiche di interazione con l'utente. L'architettura di un sistema è una sequenza di LAYER verticali mantenuta in fase di progetto e implementazione.

Layer

Si separa il dominio, che realizza l'entità dell'applicazione, dalla parte che realizza l'interazione con l'utente, la logica di presentazione, con una parte centrale di connessione, il control. Il livello di PRESENTAZIONE comprende parti per l'interfaccia utente, si astraggono il più possibile i dettagli dei dispositivi I/O per la riusabilità. Il livello di APPLICAZIONE comprende parti che provvedono a elaborare l'informazione in ingresso, a produrre risultati attesi e presentare le informazioni in uscita. Il livello delle ENTITÀ forma il modello del dominio applicativo.

Struttura: package

Tenendo conto del lavoro precedente, il primo package è quello del modello del dominio, se ben realizzato costituisce la parte entity. Si può creare un package ogni funzionalità identificata nella tabella delle funzionalità, il control. Si possono creare package per il boundary e si identificano le DIPENDENZE LOGICHE tra package.

Struttura: classi

Dopo aver definito la struttura di alto livello si dettagliano le classi di ogni package. Se le classi sono poche si può fare un unico diagramma delle classi. Sennò per ogni package si può creare un diagramma delle classi. Non introduciamo scelte di progettazione in questa fase, bisogna indicare solo le classi deducibili dal problema. Viene inserita almeno una classe che realizza funzionalità indicate nelle specifiche, se nel diagramma dei package è stata indicata una funzionalità che era stata scomposta si possono indicare le classi delle sottofunzionalità, si devono indicare le classi che rappresentano delle classi di interazione con l'utente, identificate nelle fasi precedenti.

Architettura logica: interazione

Descrive le interazioni tra le entità identificate nella parte di struttura con i diagrammi di sequenza. Evidenziano lo scambio di messaggi tra oggetti (interazioni) e l'ordine in cui i messaggi vengono scambiati. Non bisogna spingere la definizione dei diagrammi nei minimi dettagli, ma solo per descrivere il funzionamento del sistema in risposta a sollecitazioni esterne, fasi particolarmente significative, casi più critici.

Architettura logica: comportamento

Descrive il comportamento delle entità della struttura con diagrammi di stato o attività. Il diagramma di stato mostra come si comportano entità complesse a seguito di interazioni ed eventi che avvengono nel sistema. Il diagramma delle attività dettaglia i funzionamenti complessi delle entità. Non si spinge la definizione nei minimi dettagli. Lo stato di un oggetto è dato dal valore degli attributi e delle associazioni. Spesso esistono oggetti che a seconda del proprio stato rispondono in maniera diversa ai messaggi ricevuti (dispositivi spenti, in attesa, operativi, guasti, ma anche transazioni in definizione, in esecuzione, completate, fallite...). È opportuno in questi casi disegnare un diagramma di stato per l'oggetto, mostrando gli stati e gli eventi che attivano le transizioni da uno stato all'altro. A un oggetto possiamo assegnare responsabilità che comportano un insieme di elaborazioni complesse che devono essere eseguite in ordine particolare. In questi casi bisogna disegnare un diagramma di attività per l'oggetto che mostri le diverse elaborazioni da portare a termine e l'ordine di tali elaborazioni.

Definizione del piano di lavoro

Dopo aver creato l'architettura logica si può iniziare a suddividere il lavoro. Bisogna suddividere le responsabilità tra membri del team di progetto, stabilire le tempistiche per la progettazione di ogni parte, stabilire i tempi di sviluppo per tutti i sottosistemi, programmare i test di integrazione tra le parti, identificare i tempi di rilascio delle diverse versioni del prototipo, indentificare un piano per gli sviluppi futuri.

Definizione del piano di collaudo

I modelli definiti dovrebbero dare informazioni su cosa le parti del sistema debbano fare, senza molti dettagli. Il cosa fare comprende anche le interazioni. Il piano di collaudo cerca di precisare il comportamento atteso da parte di un'entità prima ancora di realizzare il progetto e la realizzazione. Si focalizza l'attenzione sulle interfacce delle entità e sulle interazioni è possibile impostare scenari dove specificare in modo già abbastanza dettagliato la risposta di una parte da uno stimolo di un'altra parte. Definire precisamente il piano di collaudo è compensato da una migliore comprensione dei requisiti, un approfondimento nella comprensione dei problemi, una definizione dell'insieme di funzionalità che ciascuna parte deve fornire alle altre per integrare tutto quello che costituisce il sistema, permette di comprendere il significato delle entità e specificarne in modo chiaro il comportamento atteso. Va concepito dal punto di vista logico, individuando comportamenti e punti critici. Può anche risultare precoce definire piani concretamente eseguibili con JUnit o NUnit. Definire un piano di collaudo eseguibile promuove uno sviluppo controllato, sicuro, consapevole del codice, perché il progettista e lo sviluppatore possono verificare subito in modo concreto la correttezza di quanto sviluppato.

Progettazione per la sicurezza

Introduzione

La sicurezza non può essere aggiunta al sistema. Deve essere progettata insieme al sistema. È anche un problema implementativo, perché le vulnerabilità sono introdotte in fase di implementazione, si può ottenere un'implementazione insicura da una progettazione sicura e viceversa.

Progettazione architetturale

L'architettura di un sistema influenza la sicurezza, se inappropriata non garantisce disponibilità, riservatezza e integrità delle informazioni. Bisogna considerare la PROTEZIONE, cioè come organizzare il sistema così che i beni critici siano protetti da attacchi esterni e la DISTRIBUZIONE, cioè come dovrebbero essere distribuiti i beni in modo da minimizzare gli effetti di un attacco andato a buon fine. Sono due problemi quasi in conflitto, perché distribuire i beni ha un costo per la protezione ed è più facile che la protezione possa fallire, ma si perde solo una parte di questi. La migliore architettura è quella a layer. I beni critici sono posizionati in basso. Il numero di layer necessari varia da applicazione ad applicazione e dipende dalla criticità dei beni da proteggere, per migliorare la protezione sarebbe bene che le credenziali di accesso siano diverse tra loro. Se è un requisito critico, si può usare un'architettura client server con meccanismi di protezione nella macchina server. La versione tradizionale ha dei limiti, con la compromissione si avremo alte perdite, i costi di recupero sono elevati, il sistema è soggetto ad attacchi DoS che sovraccaricano il server; quindi si può adottare un'architettura distribuita, il server viene replicato in punti diversi della rete, sarà più semplice il rispristino dei dati in caso di attacco e il sistema continuerà a funzionare. L'architettura più appropriata per la sicurezza può essere in conflitto con altri requisiti di applicazione, come la riservatezza dei dati di un grande database contro l'accesso veloce ai dati. Riservatezza e velocità sono in conflitto, perché i layer di riservatezza rallentano il sistema e un'architettura snella diminuisce la riservatezza. Va valutato quale requisito è prioritario, l'architettura cambierà in base a questo.

Linee guida di progettazione

Tipi di sistema diversi chiedono misure tecniche diverse per avere un buon livello di sicurezza. I requisiti influenzano pesantemente cosa è accettabile. Ci sono linee guida generali, come base per una lista di controlli da fare in validazione e per migliorare la consapevolezza degli sviluppatori. Bisogna: basare le decisioni della sicurezza su una politica esplicita; evitare un singolo punto di fallimento; fallire in modo certo; bilanciare sicurezza e usabilità; essere consapevoli dell'esistenza dell'ingegneria sociale; usare ridondanza e diversità riduce i rischi; validare tutti gli input; dividere i beni in compartimenti; progettare per il deployment; progettare per il ripristino.

Basare la sicurezza su policy

La Security Policy è un documento di alto livello, definisce cos'è la sicurezza e non come ottenerla. Non dovrebbe definire meccanismi usati per fornire e far rispettare la sicurezza. Gli aspetti della security policy dovrebbero originare da requisiti di sistema; questo è poco probabile, soprattutto se viene adottato in un rapido processo di sviluppo. I progettisti devono consultare la policy nelle decisioni di progettazione e nella loro valutazione.

Progettazione delle politiche

Devono essere incorporate nella progettazione per specificare come le informazioni possono essere accedute, quali precondizioni devono essere testate per l'accesso, a chi concedere l'accesso. Tipicamente sono rappresentate come un insieme di regole e condizioni. Devono essere incorporate in un componente chiamato Security Authority, con il compito di far rispettare le politiche all'interno dell'applicazione. A livello progettuale le politiche di sicurezza sono suddivise in: IDENTITY POLICIES, regole per la verifica di credenziali degli utenti; ACCESS CONTROL POLICIES, da applicare alle richieste di accesso e all'esecuzione di specifiche operazioni a disposizione delle applicazioni; CONTENT SPECIFIC POLICIES, da applicare a informazioni durante la memorizzazione e la comunicazione; NETWORK AND INFRASTRUCTURE POLICIES, regole per controllare il flusso e il deployment delle reti e dei servizi infrastrutturali di hosting pubblici e privati; REGULATORY POLICIES, regole a cui l'applicazione deve sottostare per essere regolamentato in un paese; ADVISOR AND INFORMATION POLICIES, regole non imposte ma consigliate per l'organizzazione e ruolo delle attività di business.

Evitare punto singolo di fallimento

È buona norma evitarli perché si eviti il fallimento di tutto il sistema. Significa non affidarsi a un singolo meccanismo per affidarsi alla sicurezza ad esempio. Viene chiamato difesa in profondità.

Fallire in modo certo

Qualche tipo di fallimento è inevitabile, quindi si dovrebbe sempre fallire in modo sicuro. Non si dovrebbero avere procedure di fall back meno sicure del sistema. Se fallisce il sistema un attaccante non deve accedere a dati riservati.

Bilanciare sicurezza e usabilità

Di solito sono in contrasto, perché per la sicurezza bisogna introdurre controlli e questo ricade sull'utente che dovrà impiegare più tempo per imparare a usare il sistema.

Consapevolezza dell'ingegneria sociale

Significa trovare modi per convincere gli utenti a rilevare informazioni riservate con l'inganno. Dipende da quanto le persone si fidano dell'organizzazione. Contrastare l'ingegneria sociale è quasi impossibile. Con una sicurezza critica le password non bastano. I log che tracciano la locazione e l'identità dell'utente con programmi di analisi del log possono essere utili ad identificare brecce nella sicurezza.

Usare ridondanza e diversità

Ridondanza vuol dire mantenere più di una versione del software e dei dati nel sistema. Vuol dire che le diverse versioni del sistema non dovrebbero usare la stessa piattaforma o essere basati sulle stesse tecnologie. Le vulnerabilità di una piattaforma non influiscono su tutte le versioni e non condurrà a un comune punto di fallimento.

Validare tutti gli input

Un attacco comune consiste nel fornire input inaspettati che causano imprevisti come il buffer overflow o la SQL injection. Per evitare questi problemi basterebbe introdurre la validazione dell'input. Tutti i controlli devono essere specificati nei requisiti, tramite la conoscenza dell'input.

Dividere i beni in compartimenti

Vuol dire organizzare le informazioni in modo che gli utenti abbiano accesso solo alle informazioni necessarie, gli effetti di un attacco sono più contenuti, non tutte le informazioni verranno probabilmente coinvolte in un attacco.

Progettazione per il deployment

Molti problemi insorgono perché il sistema non è configurato correttamente al momento del deployment. Il sistema deve essere progettato in modo da includere gli strumenti per semplificare il deployment. Il deployment coinvolge la configurazione del sistema per operare nell'ambiente, una semplice impostazione di parametri delle preferenze degli utenti, definizione di regole e modelli di business che governano l'esecuzione del software, installazione del sistema sui computer dell'ambiente, configurazione del sistema installato. Nella fase di deployment si introducono accidentalmente priorità. Il deployment è spesso visto come un problema amministrativo fuori dall'amministrazione, quindi i progettisti hanno la responsabilità di progettare per il deployment. Bisogna fornire supporti per il deployment per ridurre la probabilità che gli amministratori compiano errori. Esistono linee guida per la progettazione e il deployment: supporto per visionare e analizzare le configurazioni, minimizzare i privilegi di default, localizzare le impostazioni di configurazione, fornire modi per rimediare a vulnerabilità di sicurezza.

Progettazione per il rispristino

Bisogna sempre progettare il sistema pensando che gli errori di sicurezza possono accadere. Quindi bisogna pensare come ripristinare il sistema dopo possibili errori e riportarlo a uno stato operazionale sicuro.

Deployment del software

Si intende comprendere e definire l'ambiente operativo del software, configurare il software con dettagli ambientali, installare il software sui computer in cui opera, configurare il software con i dettagli dei computer.

Supporto per le configurazioni

Bisogna includere programmi di utilità per esaminare la configurazione corrente del sistema. Questi programmi di solito mancano nella maggior parte dei sistemi software. Gli utenti sono frustrati nella difficoltà di trovare dettagli di configurazione (quando occorre visionare diversi menu si va ad errori e omissioni). In fase di visualizzazione si dovrebbero evidenziare impostazioni critiche per la sicurezza.

Minimizzare i privilegi di default

Il software deve essere progettato affinché la configurazione di default fornisca i minimi privilegi essenziali. Così vengono eliminati i danni di un possibile attacco.

Localizzare impostazioni di configurazione

Quando si progetta il supporto per le configurazioni del sistema bisognerebbe assicurarsi che ogni risorsa che appartiene alla stessa parte del sistema venga configurata nella stessa posizione. Se le informazioni di configurazione non sono localizzate è facile dimenticarsi di farlo, può capitare di non essere a conoscenza dell'esistenza di meccanismi per la sicurezza inclusi nel sistema, se tali meccanismi presentano configurazioni di default si può essere esposti ad attacchi.

Rimediare a vulnerabilità

Bisogna includere meccanismi diretti per aggiornare il sistema, riparare le vulnerabilità di sicurezza che vengono scoperte. Questi potrebbero includere verifiche automatiche per aggiornamenti di sicurezza e download di questi appena sono disponibili. Gli aggiornamenti devono coinvolgere centinaia di PC su cui il software è aggiornato.

Testare la sicurezza

Il test di un sistema gioca un ruolo chiave nel processo di sviluppo del software e dovrebbe essere eseguito con molta attenzione. L'area dei test della sicurezza è quella più trascurata nello sviluppo, perché non c'è tempo, non c'è il know how, non ci sono gli strumenti o manca la comprensione dell'importanza. Il test della sicurezza è in lavoro lungo, tedioso, molto più tedioso di test funzionali svolti normalmente. Coinvolge diverse discipline (test simili per il team testing e test volti a rompere il sistema, il black box testing e il white box testing).

Black box testing

Si assume di non conoscere l'applicazione e i tester affrontano l'applicazione come un attaccante: si indaga sulle informazioni della struttura interna e si applicano un insieme di tentativi di violazione su queste informazioni. Si possono impiegare vari strumenti, tool che permettono di scandagliare le porte, ma vengono anche fatti test di livello infrastrutturale (errori di confiurazione, falle di sicurezza delle macchine virtuali, difetti dei linguaggi di programmazione).

White box testing

Si assume di conoscere l'applicazione. I tester possono sapere qualunque cosa della configurazione e del codice, cercano possibili debolezze. Gli strumenti sono diversi da quelli del black box test, sono tool di debugging, di solito si parla di corse critiche o memory leak.

Capacità di sopravvivenza

È la capacità di continuare a fornire servizi essenziali a utenti legittimi mentre il sistema è sotto attacco o dopo che parti del sistema sono state danneggiate come conseguenza di un attacco o un fallimento. È una proprietà del sistema, non dei singoli componenti. È un lavoro critico perché né dipende la vita e l'economia delle infrastrutture. Analizzare e progettare la capacità di sopravvivenza è parte del successo di ingegnerizzazione dei sistemi sicuri, la disponibilità dei servizi è l'essenza della sopravvivenza. Il Survivable Analysis Systems è un metodo di analisi che valuta la vulnerabilità del sistema e supporta la progettazione di architetture che promuovono la sopravvivenza del sistema. Le strategie sono: RESISTENZA, evitare problemi costruendo la capacità di respingere attacchi; IDENTIFICAZIONE, individuare problemi costruendo all'interno del sistema le capacità di riconoscere attacchi e fallimenti; RIPRISTINO, tollerare problemi costruendo all'interno del sistema le capacità di fornire servizi essenziali durante un attacco e ripristinare le complete funzionalità dopo l'attacco. Bisogna capire il sistema (riesaminare gli obiettivi, i requisiti e l'architettura), identificare i servizi critici (quali mantenere e i componenti per farlo), simulare gli attacchi (identificare casi d'uso e scenari di attacco) e analizzare la sopravvivenza (identificare i componenti essenziali a rischio e le strategie di sopravvivenza basate su resistenza, identificazione e ripristino). Aggiungere le tecniche di sopravvivenza costa soldi, spesso le aziende sono molto riluttanti a investire su questo se non hanno già subito perdite, è comunque buona norma investire nella sopravvivenza prima che dopo, l'analisi non è inclusa nella maggior parte dei processi di ingegnerizzazione del software e sembra probabile che questo tipo di analisi sarà sempre più utilizzato.

Progettazione

Introduzione

L'obiettivo è che attraverso dei raffinamenti si passi dall'architettura logica all'architettura del sistema. Bisogna considerare gli aspetti vincolanti che sono stati ignorati nelle fasi precedenti. Non solo si individua la soluzione al problema, ma si descrivono i motivi per cui si adotta.

Progettazione architetturale

Gli ingegneri devono prendere decisioni che influenzano il sistema. C'è un'architettura applicativa generica che può essere usata per il sistema che sto progettando? Come sarà distribuito il sistema tra più processori? Quali stili sono adatti al sistema? Qual è l'approccio usato per strutturare il sistema? Come saranno scomposte in moduli le unità strutturali del sistema? Quale strategia sarà usata per controllare l'operato delle unità del sistema?

Requisiti non funzionali

L'architettura influenza le prestazioni, la robustezza, la distribuibilità, la manutenibilità di un sistema. La struttura dell'architettura tipicamente è condizionata da quale applicazione si vuole realizzare e dai requisiti non funzionali. Se le PRESTAZIONI sono un requisito critico l'architettura dovrebbe essere progettata localizzando le operazioni critiche all'interno di un piccolo numero di componenti, minimizzando le comunicazioni possibili tra essi. Bisogna definire comportamenti grandi per ridurre la comunicazione. Se la PROTEZIONE DATI è un requisito critico l'architettura deve essere progettata con una struttura stratificata collocando le risorse più critiche nello strato più interno e protetto. Questo porta a dover definire una struttura con un alto livello di convalida di protezione dei dati. Se la SICUREZZA è un requisito critico bisogna progettarla affinché le operazioni relative siano collocato in un piccolo insieme di componenti e si riducano i costi e i problemi di convalida della sicurezza, si possono fornire sistemi di protezione correlati. Bisogna definire anche componenti grandi per localizzare le operazioni. Se la DISPONIBILTÀ è un requisito critico, bisogna progettare l'architettura per avere componenti ridondanti e che sia possibile aggiornarli e sostituirli senza frenare il sistema. Se la MANUTENIBILITÀ è un requisito critico, bisogna usare componenti piccoli, atomici, autonomi, modificabili velocemente, i produttori di informazione separati dai consumatori e le strutture dati condivise devono essere evitate. Questo porta a sviluppare componenti di piccole dimensioni. Ci sono conflitti potenziali tra queste architetture e se ci sono requisiti in contrasto critici bisogna trovare un compromesso.

Scelta dell'architettura

Deve basarsi su: architettura logica dell'analisi del problema, trade-off dei requisiti non funzionali, tipologia di applicazione che si intende sviluppare, adozione di pattern architetturali. Non esiste un'architettura ideale sempre utilizzabile. È necessario usare stili architetturali diversi per parti diverse del sistema al fine di soddisfare i vincoli imposti dai requisiti. L'adozione di pattern architetturali può aiutare a trovare il giusto compromesso tra tutte le forze in gioco. L'uso di una tecnologia non è sempre neutro. Potrebbe risultare vantaggioso scegliere tecnologie in fase di progettazione per legare il progetto alla specifica tecnologia. Se scegliamo la tecnologia in fase di progettazione dobbiamo specificarlo chiaramente con un'analisi costi/beneifici. Vanno studiate le parti di tecnologia adottata affinché sia possibile inserirle nei diagrammi di progettazione.

Blackboard

Aiuta a strutturare quelle applicazioni in cui vengono applicate strategie di soluzione non deterministiche. I diversi sottosistemi condividono le stesse conoscenze sulla blackboard per costituire una soluzione approssimata o parziale.

MVC

Divide le applicazioni in tre distinte parti: MODEL, gestisce i dati; CONTROLLER, manipola i dati; VIEW, mostra i dati.

Layer

Aiuta a strutturare le applicazioni che possono essere scomposte in gruppi di sottoattività in cui ciascun gruppo si trova a un ben definito livello di astrazione.

Client/Server

Aiuta a strutturare un'applicazione come insieme di servizi forniti da uno o più server e un insieme di client che usa tali servizi. Il THIN CLIENT lascia tutto a carico del server, il FAT CLIENT dà al client la logica applicativa e lascia al server la gestione dei dati, il PRESENTATION è un client/server a 3 livelli (Data Server, Application Server, Client).

Broker

Può essere usato per strutturare sistemi distribuiti con disaccoppiamento tra diversi sottosistemi che comunicano attraverso la remote server invocation. È responsabile della coordinazione delle comunicazioni come inoltro richieste, invio risposte ed eccezioni.

Pipe & filters

Aiuta a strutturare quelle applicazioni che processano flussi di dati. Ogni passo del processo è incapsulato in un filtro e i dati attraversano una pipe di filtri. Variando l'ordine dei filtri possiamo ottenere tipi diversi di sistemi.

Progettazione di dettaglio

Definisce il dettaglio dell'architettura del sistema in tre viste: struttura, interazione e comportamento. Per realizzare un sistema funzionante bisogna considerare GUI, DB, Framework, librerie, componenti, modifiche al modello per avere software estensibile e modulare. È compito della progettazione di dettaglio identificare e definire altre classi in accordo alla specifica architettura scelta. I modelli prodotti dall'analisi devono essere estesi al fine di progettare i quattro layer principali che compongono il sistema: APPLICATION LOGIC e controllo degli altri componenti; PRESENTATION LOGIC, gestione dell'interazione con l'utente a livello logico; DATA LOGIC, gestione dei dati che il sistema deve manipolare; MIDDLEWARE, gestione dell'interazione con sistemi esterni, con la rete e tra sottosistemi. Bisogna modificare i modelli di analisi affinché si definiscano in dettaglio classi e relazioni, si supportino caratteristiche specifiche, si riusino classi o componenti disponibili, si migliorino le prestazioni, si migliori il supporto alla portabilità. Bisogna essere indipendenti dal linguaggio di programmazione, DBMS, OS, hardware, a meno che non sia un requisito non funzionale o si è esplicitamente scelto di legarsi a una tecnologia nella progettazione architetturale.

Architettura: struttura

Bisogna definire i tipi di dato non definiti in precedenza, la navigabilità delle associazioni tra classi, strutture dati necessarie per l'effettiva implementazione del sistema, operazioni non emerse durante l'analisi del problema, eventuali nuove classi per il corretto funzionamento del sistema. Attenzione alla presenza di sistemi esterni, se nella tabella era stato individuato un problema nel livello di protezione e il sistema esterno non risulta avere il livello di sicurezza minimo richiesto, bisogna usare il pattern adapter: si ingloba il sistema esterno in una struttura, si progetta la struttura affinché soddisfi i livelli minimi di sicurezza richiesti. Attenzione se ci si vincola ad una tecnologia, bisogna analizzare il livello di protezione offerto e se tale livello non è il minimo richiesto bisogna progettare specifiche parti del sistema per prevenire i buchi di sicurezza, usare il pattern adapter. Si applicano i design pattern per realizzare software di qualità, estensibile e modulare. Fare attenzione al dependency inversion principle. Disaccoppiare i layer del sistema porta molti vantaggi, il design for change, si può cambiare l'aspetto grafico variando la tecnologia realizzativa senza modificare l'application logic, si possono inserire nuove funzionalità con impatto minimo sul sistema.

Architettura: interazione

È necessario ridefinire i protocolli d'interazione definiti in analisi, tenendo conto delle nuove entità emerse in progettazione, progettare accuratamente i protocolli di interazione verso i sistemi esterni, definire nuovi protocolli di interazione tra le classi che sono state introdotte nella progettazione.

Architettura: comportamento

È necessario definire gli algoritmi che implementano le operazioni complesse in modo chiaro e preciso avvalendosi di diagrammi delle attività, dettagliare i diagrammi di stato/attività definiti nella fase precedente, eventualmente aggiungere diagrammi di stato/attività per le nuove entità emerse in questa fase.

Progettazione della persistenza

È un fattore cruciale nello sviluppo di un sistema. Il progettista deve valutare attentamente i vincoli imposti dai requisiti funzionali, la tipologia di accesso ai dati, la frequenza di accesso ai dati, la criticità e consistenza dei dati e dovrà scegliere la tecnica migliore di persistenza. Per ogni sistema bisogna valutare attentamente quale strategia dà il miglior bilanciamento tra vincoli e forze in gioco nel sistema. Non è detto che adottare un (R)DBMS sia sempre la risposta corretta, ad esempio per scrivere i log occorre memorizzarli su un file. Possiamo usare il DB quando abbiamo gestionali con un considerevole numero di dati anche di natura eterogenea, dati che cambiano molto spesso e devono essere costantemente aggiornati, dove la perdita di modifiche può essere un problema, la necessità di ripristino di versioni precedenti a seguito di un malfunzionamento. L'output di questa fase può essere rappresentato dallo schema E-R del DB che dovrà supportare l'applicazione, il formato dei file dovranno essere scritti e letti dall'applicazione. Sarebbe bene che sia nel caso del DB che di file ci fosse una piccola analisi del rischio per capire se il DB è protetto in modo adeguato e i file necessitano di meccanismi di protezione. Il punto di partenza dell'analisi sono i livelli di protezione e privacy richiesti per i diversi dati che saranno memorizzati.

Progettazione del collaudo

La base di partenza di questa attività è il piano di collaudo sviluppato nell'analisi. Dopo la progettazione di dettaglio è possibile scrivere i test unitari di ciascuna classe. Successivamente vanno progettati con cura anche i test di integrazione del sistema. L'output di questa attività è rappresentato dalla suite completa di test unitari e di integrazione.

Progettazione per il deployment

Seguiremo le linee guida viste in sicurezza: includere un supporto per visionare e analizzare le configurazioni, minimizzare i privilegi di default, localizzare le impostazioni di configurazione, fornire modi per rimediare a vulnerabilità di sicurezza.

Diagramma dei componenti e di deployment

Diagramma dei componenti

In UML 2.0 il concetto di componente si è evoluto. Specifica un contratto formale di servizi offerti e richiesti in termini di interfacce. Questo concetto è legato alla STRUTTURA COMPOSITA che viene impiegata per rappresentare le parti interne del componente. Un componente è tipicamente specificato da uno o più classificatori e può essere implementato da uno o più artefatti. Gli internals sono inaccessibili solo attraverso le interfacce. Da UML 2.0 è una specializzazione della metaclasse class, un componente può avere attributi e metodi, una struttura interna, porte e connettori, inoltre cambia in UML anche l'icona del componente. L'interfaccia fornita e richiesta devono essere compatibili a livello di tipo (attributi e associazioni) e vincoli sul comportamento (operazioni, eventi). I classificatori interni che realizzano un componente possono essere mostrati innestandoli nel componente o in modo esplicito tramite la dipendenza di realization. La realization è una dipendenza tra due insiemi di elementi di modellazione, uno rappresenta la specifica e l'altro l'implementazione. Per un componente la realization definisce i classificatori che realizzano il contratto offerto dal componente stesso in termini di interfacce e offerte richieste. Le parti interne sono collegate tra loro o connesse a porte sul confine del componente. I delegated connector sono usati per esporre servizi di una parte all'esterno del container. UML 2.0 permette di connettere alla stessa porta più interfacce. Se in UML 1.x un subsystem è un tipo di package in UML 2.0 è un tipo di componente, quindi si può specificare per un subsystem le interfacce richieste e fornite, per evidenziare le relazioni con altri subsystem. Il diagramma dei componenti deve essere impiegato negli stadi finali della progettazione. Il diagramma rispecchia la struttura che deve avere il codice e rappresenta l'architettura del sistema. Può essere pensato come stadio finale dell'evoluzione dell'architettura logica che in fase di analisi viene rappresentata attraverso il diagramma dei package.

Struttura composita

Il diagramma di struttura composita ha l'obiettivo di rappresentare la struttura interna di un classificatore, inclusi i punti di interazione utilizzati per accedere alle caratteristiche della struttura. Serve per scomporre gerarchicamente un classificatore, ne mostra la struttura interna, separa l'interfaccia dalla struttura interna, descrive i ruoli di diversi elementi della struttura gerarchica e le interazioni. Permette al progettista di prendere un oggetto complesso e spezzarlo in parti più piccole e semplici. È uno strumento di zoom per gestire la complessità di rappresentazione.

Package e struttura composita

La differenza sta nel fatto che i primi rappresentano un raggruppamento logico al momento dell'analisi. Le seconde si riferiscono a quello che succede durante l'esecuzione. Le strutture composite sono adatte a rappresentare i componenti e le loro parti e sono usate nei diagrammi dei componenti.

Diagramma di deployment

Documentano la distribuzione fisica di un sistema, mostrando i pezzi di software in esecuzione su macchine fisiche, mostrano i collegamenti che permettono la comunicazione fisica tra pezzi hardware e le relazioni tra macchine fisiche e processi software con l'indicazione di vari punti in cui viene eseguito il codice. L'ARTIFACT rappresenta una porzione fisica di informazioni prodotta o usate nel ciclo di sviluppo del software, viene usato il <<manifest>> che illustra elementi di modellazione usati per generare l'artefatto. Il NODE è un'unità su cui risiedono o sono eseguiti componenti artefatti, comunicano tramite CommunicationPath, l'allocazione degli artefatti su un nodo si rappresenta con <<deploy>> tra nodo e artefatto. Il DEVICE è la risorsa fisica con capacità elaborative dove si possono allocare gli artefatti per l'esecuzione. Il MANIFEST è la relazione di dipendenza che illustra elementi di modellazione usati nella costruzione o generazione di un artefatto. Il nodo rappresenta qualsiasi cosa su cui si può eseguire un lavoro. È una risorsa su cui gli artefatti possono essere allocati per l'esecuzione. L'Execution Environment è un nodo che offre l'ambiente per l'esecuzione di specifici tipi di componenti allocati su di esso (<<OS>>, <<databasesystem>>). Il Deployment Specification è un insieme di proprietà che determinano i parametri di esecuzione di un artefatto allocato su un nodo.

Progettazione concettuale (E-R)

Il primo passo

Rappresentiamo i requisiti del SI e tramite la progettazione concettuale abbiamo lo schema concettuale. Lo rappresentiamo attraverso la progettazione logica con lo schema logico e la progettazione fisica fornisce lo schema fisico.

Raccolta dei requisiti

I requisiti devono essere acquisiti, le fonti sono gli utenti (interviste o documentazione apposita) e la documentazione esistente (normative, regolamenti interni, realizzazioni preesistenti), modulistica. La raccolta dei requisiti è un'attività difficile non standardizzabile.

Interagire con gli utenti

Bisogna considerarla con molta attenzione, perché utenti diversi possono fornire informazioni diverse, a livello più alto hanno una visione più ampia ma meno dettagliata. In generale risulta utile effettuare spesso verifiche di comprensione e coerenza, verificare anche per mezzo di esempi, richiedere definizioni e classificazioni, far evidenziare aspetti essenziali rispetto ai marginali.

Requisiti: documentazione descrittiva

Le regole generali sono lo scegliere corretto del LIVELLO DI ASTRAZIONE, STANDARDIZZARE la struttura delle frasi, SUDDIVIDERE le frasi articolate, SEPARARE le frasi sui dati da quelle sulle funzioni. Per evidenziare meglio i concetti espressi è opportuno costruire un GLOSSARIO DEI TERMINI, individuare OMONIMI e SINONIMI e unificare i termini, rendere esplicito il RIFERIMENTO FRA TERMINI, riorganizzare le frasi per concetti.

Glossario dei termini, omonimi e sinonimi

Raramente i requisiti espressi in linguaggio naturale non hanno ambiguità. È frequente il caso di OMONIMI (stesso termine per concetti diversi) e SINONIMI (termini diversi usati per descrivere lo stesso concetto). Per rappresentare i concetti più rilevanti è conveniente usare il GLOSSARIO DEI TERMINI, descrive brevemente il concetto, indica gli eventuali sinonimi, crea relazioni tra concetti del glossario stesso.

Ristrutturazione dei requisiti

Per semplificare le analisi successive, è utile riformulare i requisiti: eliminare le omonimie, usare un termine univoco per ogni concetto, riorganizzare frasi raggruppandole in base al concetto a cui si riferiscono.

Dai concetti allo schema E-R

Va sempre ricordato che un concetto non è di per sé un'entità, un'associazione, un attributo o altro, DIPENDE DAL CONTESTO. Si rappresenta come ENTITÀ il concetto con proprietà significative e descrive oggetti con esistenza autonoma; come ATTRIBUTO un concetto semplice senza proprietà; come ASSOCIAZIONE se il concetto correla due o più concetti; come GENERALIZZAZIONE/SPECIALIZZAZIONE se il concetto è un caso più generale/particolare di un altro.

Strategie di progettazione

Per progetti complessi è opportuno adottare un modo specifico di procedere, una strategia di progettazione, i casi notevoli sono: TOP-DOWN, si parte da uno schema iniziale astratto ma completo che viene raffinato fino allo schema finale, non è inizialmente necessario specificare i dettagli, ma richiede una visione globale del problema, a volte non è semplice averla; BOTTOM-UP, si suddividono le specifiche in modo da sviluppare semplici schemi parziali ma dettagliati che vengono integrati tra loro, permette una ripartizione delle attività, ma richiede anche una fase di integrazione; INSIDE-OUT, si sviluppa lo schema a macchia d'olio, cioè si parte dai concetti più importanti e si espandono aggiungendo quelli correlati e così via, non richiede passi di integrazione, ma ad ogni passo richiede di esaminare tutte le specifiche per trovare i concetti non ancora rappresentati. Di solito si usa una strategia ibrida, si individuano i concetti principali e si realizza uno SCHEMA SCHELETRO che contiene i concetti più importanti, poi questo si può DECOMPORRE, si RAFFINA, si ESPANDE, si INTEGRA.

Qualità di uno schema concettuale

Lo schema E-R deve essere verificato affinché risponda ha i requisiti di: CORRETTEZZA, non devono essere presenti errori sintattici o semantici; COMPLETEZZA, tutti i dati di interesse devono essere specificati; LEGGIBILITÀ, riguarda aspetti anche prettamente estetici dello schema; MINIMALITÀ, bisogna capire se ci sono elementi ridondanti nello schema, potrebbe non essere un problema, ma una scelta di progettazione che favorisce certe operazioni.

Metodologia basata sulla strategia mista

Nell'ANALISI DEI REQUISITI si analizzano i requisiti e si eliminano le ambiguità, si costruisce un glossario dei termini e si raggruppano i requisiti; nel PASSO BASE si definisce uno schema scheletro con i concetti più rilevanti; nel PASSO DI DECOMPOSIZIONE (se necessario e appropriato) si decompongono i requisiti con riferimento a concetti nello schema scheletro; nel PASSO ITERATIVO (da ripetere se non si è soddisfatti) si raffinano i concetti presenti sulla base delle loro specifiche, si aggiungono concetti per descrivere specifiche non descritte; nel PASSO DI INTEGRAZIONE (se si è decomposto) si integrano vari sottoschemi in uno schema complessivo, facendo riferimento allo schema scheletro; nell'ANALISI DI QUALITÀ (ripetuta e distribuita) si verifica la qualità dello schema e si modifica.

Riassumendo

La FASE DI ANALISI DEI REQUISITI è fondamentale per poter progettare una base di dati che rispetti i requisiti. Mancando la possibilità di standardizzarla, tale fase si avvale necessariamente di regole di buon senso e di una serie di strumenti che riducono il rischio di commettere errori grossolani, oltre a costruire una valida documentazione. Per la PROGETTAZIONE PER LO SCHEMA E-R sono possibili diverse strategie, di cui quella MISTA è senz'altro la più diffusa e adeguata anche nel caso di progetti estremamente complessi.